KUMA FORTH

FOR

THE

MSX MICRO-COMPUTER

ISBN NO 07457-0006-3

This program is supplied in the belief that it operates as specified, but Kuma Computers Ltd. (the company) shall not be liable in any circumstances whatsoever for any direct or indirect loss or damage to property incurred or suffered by the customer or any other person as a result of any fault or defect in goods or services supplied by the company and in no circumstances shall the Company be liable for consequental damage or loss of profits (whether or not the possibility thereof was seperately advised to it or reasonably foreseeable arises from the use or performance of such goods or services.

Published by:- Kuma Computers Ltd.,
12 Horseshoe Park,
Pangbourne,
Berks RG8 7JW

Telex 849462 Tel 07357 4335

Copyright Licence

"the Software" the software program or programs supplied with this Licence

"the Licenser" Kuma Computers Ltd

In consideration of the lump sum payment by the Customer the Licenser hereby grants to the Customer a non-exclusive non-transferable Licence to use the Software on the following terms and conditions.

1 The Licence shall commence on receipt of the Software by the Customer and shall continue thereafter unless terminated in accordance with the terms hereof.

2 The Licence authorises only the Customer to use the Software only on equipment owned or used by him.

3 The Software and the copyright in and title to all industrial property rights therein are and shall remain the property of the Licenser.

4 The Customer shall not donate market lend or otherwise dispose of the Software to any other person or company.

5 The Customer shall not assign sublet or otherwise transfer this Licence or the Software by operation of law or otherwise in whole or in part directly or indirectly and any purported assignment of this Licence by the Customer shall be void.

6 The Customer shall not make copies of the Software in machine readable or printed or other form except that such copies shall be for his personal use and provided that such copies will be kept under his close personal control and that no more than five such copies shall be in existence at any one time.

7 The Licenser may terminate this Licence forthwith upon notice if the Customer neglects or fails to perform or observe any of the terms hereof.

8 Immediately upon termination of this Licence for whatever reason the Customer shall return the Software to the Licenser and shall not have the right to retain any copies of any part thereof by any means.

9 The Licenser shall not be liable for damages or consequential damages directly or indirectly arising out of the or in connection with the delivery use or performance of the Software.

10 If any of the conditions or parts thereof of this Licence are held to be invalid inoperative or unenforceable by any Court they are to that extent to be deemed omitted. The Law of this Licence shall be the Law of England.

CONTENTS

ABOUT THIS GUIDE

This booklet is divided into four sections. The first of these contains a brief introduction to Forth and description of the features of the implementation followed by operating instructions. The latter should be read before the software is used. The other three sections are entitled Using Forth, Forth Techniques and Glossary.

Using Forth is an introduction to the Forth language. Since Forth is an interactive language this section is built round examples which can be executed simply by typing them at the keyboard. The section is tutorial in nature and concludes with a description of the editing facilities.

Forth Techniques is a much more detailed guide to Forth programming. Several different aspects of the language are covered in some depth and the presentation is a compromise between an introduction and a source of reference. This section also details floating point arithmetic.

The Glossary is a description of the words of the core language. It should be stressed that it is not necessary for the user to know all these words to make effective use of Forth. As a self defining language Forth provides a wide range of facilities, selective use of which can be made as the starting point for any particular application. As new words are added by the user the language comes to reflect the application rather than the reverse.

After reading the operating instructions most users will want to begin with the section Using Forth and then move on to Forth Techniques. Forth is an easy language to use but is likely to contain many unfamiliar ideas to start with. The different approaches used such as postfix notation, stacks and close integration with hardware are none the less more unusual than difficult. They may take a little time to grasp but the effort should be well worthwhile.

Introduction to Forth

Forth is a fully structured self extending language. It is a typeless language offering the user complete freedom to design and manipulate data structures of arbitrary complexity. It allows the user full access to all machine functions including those controlling the operation of the language itself. Programming in Forth consists of extending the language in ways defined by the user. The results are indistinguishable from the core language.

Operators in Forth are called words and are of two types known as primitive and secondary words. Primitive words are object code routines which contain Z80 machine code instructions. Secondary words contain either threaded code or data such as constants, variables and arrays. Threaded code, which is the result of compilation in Forth, consists of lists of addresses of other words to be executed in sequence like a series of subroutines. To the user, though, primitive and secondary words behave in the same way.

Words are contained in a dictionary which is a linked list so that each word contains the address of its predecessor in the dictionary. The dictionary is divided into vocabularies, each of which contains words associated with a particular function. General purpose words are found in the core vocabulary. New words are added to the dictionary using defining words which compile the language. A new word is defined as a sequence of existing words.

Processes communicate through stacks (last in first out lists). One stack, called the data stack, parameter stack or where there is no ambiguity just the stack, is explicitly available to the user for the manipulation of numbers and other temporary data. Another stack, called the return stack, does not require the intervention of the user but can also be used for data.

A virtual memory system organises available mass storage into blocks of a fixed number of bytes each. When referenced by number a block is automatically read into a buffer in the random access memory. An important use of these blocks is for the storage of text representing new Forth words and programs and for this purpose blocks are grouped into screens of 1024 characters. The text can be edited using text editor words contained in an editor vocabulary.

Forth differs from other languages because the wordset is chosen by the user. Starting from an initial general purpose set new and progressively more powerful words are built up interactively and in small steps. The resulting wordset reflects the users application and can be as specialised as required. There are as few restrictions as possible and the aim is to make the most efficient use of hardware rather than mask its existence.

Implementation Notes

The implementation conforms to the standard FIG FORTH model version 1.1. The character set is the ASCII 128 character set but other characters may be included as data or in strings.

By convention letters used in the names of Forth words are always in the upper case but lower case letters are not translated into the upper case, to overcome this it is suggested that you set the Shift Lock. Each word and number in an input line must be separated from others by at least one space and no spaces should be left in the middle of a word name of number. No action will be taken until a line is completed with a carriage return except echoing characters to the screen.

The language itself recognises only backspace, and return as control keys; all others are treated as data. A primitive word KEY allows this arrangement to be modified for particular applications such as a screen editor. See later for more details.

A table of error messages appears over the page. The error message numbers can be replaced with text by executing 1 WARNING! providing that error message on screens 4 and 5 are resident.

Error message	0	Unrecognised input
	1	Empty stack
	2	Dictionary full
	3	Has incorrect address mode
	4	Is not unique
	6	Disc range or Tape range
	7	Full stack
	8	Disc error or Tape error
	17	Compilation only; use in definition
	18	Execution only
	19	Conditionals not paired
	20	Definition not finished
	21	In protected dictionary
	22	Use only when loading
	23	Off current editing screen
	24	Declare vocabulary

The error message numbers can be restored by executing 0 WARNING! Users should not be inhibited by error message 1 Empty stack. This error is easily made and often made deliberately by Forth users to check that the stack is actually empty at a particular time.

OPERATION

Loading the Language

Before the language can be loaded, space must be created in the MSX store. This may be done using the CLEAR statement. After this the language may be loaded using the BLOAD command.
The loading procedure is therefore:-
(i) CLEAR 200,&H87FF (RETURN)
(ii) BLOAD "KFORTH",R (RETURN)

When this has been successfully achieved the heading title "KUMA FORTH (c) D. Williams, Z80 FIG FORTH 1.1" will appear together with a cursor indicating that the machine is waiting for input.

It should be noted that the resident BASIC and the FORTH co-exist in the machine. It is possible to exit back to the BASIC by using the command BYE and back again to the FORTH using the command DEFUSR=&H8800:A=USR(0).

Since FORTH is an interactive language it is ready for use as soon as it is loaded. The tutorial section 'Using FORTH' may be read at this point if the user wishes to try out the language before reading about the tape and printer facilities.

Using a Printer

An Epson MX80 F/T printer driver is installed and is enabled by storing a non zero value in the variable EPRINT. This can be done by executing 1 EPRINT ! or by control P. The printer is turned off by 0 EPRINT! or a second control P. When the printer

is enabled, everything output to the screen is echoed to the printer. This is a very convenient way of producing listings and other outputs. If a different form of output control is required or a different printer driver needed then a user supplied subroutine may be installed. This should be placed in the dictionary starting at the dictionary pointer given by HERE. Space should be reserved in the dictionary using ALLOT; 30 ALLOT reserves 30 bytes. The subroutine should be called from 9FC6 hex where the address after the call instruction CD must be altered appropriately. The character to be output is supplied in the A register in Sharp normal code and the values of the IX and IY registers must be preserved. Any other register may be used and the subroutine must end with the return instruction C9.

Tape Operation

(N.B. for successful operation of the tape interface provided it is necessary that a tape recorder with a 'remote' facility be used.)

The tape may be used to store programs and data, however, rather than providing separate files on tape such as in BASIC, FORTH considers the tape to be an extension of the memory inside the machine. To achieve this the tape is divided up into numbered blocks each capable of storing 1024 bytres of data. Each block contains a header of the form SCRN£nnn (where nnn is the block number), the 1k body and a checksum.

Before a tape can be used it must first of all be formatted like a disc. To do this put a blank tape in the recorder and press the record button. Then execute the word FORMAT. This will cause the machine to write 16 blank files onto the tape. The system will return with the word OK when this has been completed. If a different number of files is required (for instance if a longer tape is used) the variable BLOCKS/TAPE may be changed using a sequence such as 32 BLOCKS/TAPE! for 32 blocks before FORMAT is exectued.

Buffers are provided in store which are used for the temporary storage of blocks. When the user wishes to access a block, he specifies it by number. If the appropriate block is not already resident in a buffer, it will be automatically fetched from tape and placed in one. Once in a buffer the block may be read or modified just like normal memory. If any changes are made to the block it is marked as updated so that later on these changes can be rewritten back onto tape. The blocks are most often used for the storage of text consisting of FORTH definitions and programs but can be used for the storage of any kind of data. Blocks that are used for text are normally referred to as screens.

This form of storage mechanism implements the virtual memory system described in the next section.

The language tape supplied, in fact, contains 8 blocks situated immediately after the language file itself. As an example of how to use the tape system these can be examined as follows.

With blocks 4 and 5 resident try 1 WARNING !. This will replace the error message numbers with messages on screens 4 and 5. The numbers can be restored by 0 WARNING !.

Updating blocks may be achieved in the following manner. Assuming that screen 1 from the language tape is available (load it if necessary) type 1 LIST UPDATE. Type STATUS and block 1 will be shown as updated. Rewind and remove the language tape and insert the newly formatted text tape. Type SFLUSH. A prompt will appear to rewind the tape (not necessary on this occasion) and PLAY. The tape will be searched until; the header of the block to be updated is found. Press STOP when prompted then press record followed by any key. The blank data part of the block will be overwritten with a copy of the block on the language tape. STATUS will show block 1 still resident but no longer updated.

This covers all the essential elements on using the tape virtual memory system. Usually only one tape is involved and the block being saved has been created by the user using the editor commands which automatically mark the block in question as saved using SFLUSH. Several blocks can be in an updated form at one time and saved using SFLUSH. A block can be overwritten on tape any number of times allowing a large program covering several blocks to be developed and edited in small steps. Note that if the buffers become full the user will be prompted to save any updated screens; they will not be lost. Always use SFLUSH at the end of a session to ensure that any updated screens or blocks are written to tape.

There is one limitation with the tape system. In order to allow the sequential reading of blocks without need for the user to press a key for every block, the system assumes that the tape transport is in play mode. This is normally the case. However, it is possible that if the buffers are full when a block is to be read and the next one to be used is marked updated, then the new block can be read. The problem this causes is that at the end of rewriting the system will assume that the tape is in play mode when it starts to search for the new block header. However the tape will still be in record mode and left unchanged would overwrite data on the tape.

This may be overcome in two ways. The best way is to keep an eye on the status of the buffers using STATUS and to regularly use SFLUSH (you ought to be doing this anyway!). Alternatively recognise this when it happens (detected by the fact that the system prompts you to record when you are expecting it to read) and to ensure that you manually press the STOP key after the recording has taken place and then press the play. There is a gap between files with a leader tone so as long as you stop the recording within a few seconds there should be no problem.

Tape Virtual Memory System

The object of the tape virtual memory system is to make a tape, as far as possible, an extension of the random access memory. The tape is divided into numbered blocks, each of 1024 bytes or 1K. The blocksd are most often used for the storage of text consisting of Forth word definitions and programs but can be used for the storage of any type of data. A block used for the storage of text is usually called a screen.

When a block is referenced the user is prompted to load it from tape and it is stored in an area of the random access memory called a buffer. There are six buffers each capable of holding one block. Once a block is resident in a buffer its contents can easily be referred to and altered by a program. If the contents are altered then the block is marked as updated. Instead of the explicit command to save a particular block the word FLUSH will cause the user to be prompted through the process of saving all the blocks that have been updated. A block is saved by overwriting the existing version on tape so that the blocks on tape are always current.

Each tape used for the storage of blocks must first be formatted. The word FORMAT writes sixteen blank 1024 byte blocks on a tape numbered from one. The number of blocks written can be alterd by changing the value of the variable BLOCKS/TAPE, which is initially sixteen. Each block is represented on tape as a numbered header followed by 1024 bytes of data. The header is used to locate the block for both loading and saving. When a block is saved in updated form only the data part, which is found by locating the header, is overwritten.

A block is accessed by the sequence n BLOCK which will prompt the user to load block number n from tape it is is not already in a buffer. The adress of the first byte of the block will be left on the stack and any byte in the block can then be accessed by adding an offset to this address.

Whenever the contents of a block are modified the word UPDATE sets the sign bit in the status cell associated with the buffer which contains the most recently referenced block. UPDATE should be included in the definition of any word which modifies the contents of a block.

An important use of blocks is for the storage of source text allowing word definitions and programs to be modified and recompiled when required. Blocks containing source text, called screens, are displayed using n LIST. The test can be edited using the words of the EDITOR vocabulary. Any editor word which alters text on a screen will automatically mark it as updated.

Two words, SFLUSH and SCOPY, replace the more usual Forth words FLUSH and COPY and have been designed for tape systems. SFLUSH prompts the user through the process of saving any updated blocks on tape leaving them stored in the buffers but no longer marked

as updated. This is more convenient than FLUSH which removes blocks from the buffers so that they must be reloaded when required again. The sequence 7 12 SCOPY will alter the number of screen 7 in the buffers to 12 and mark it as updatd so that it will be written to block 12 on tape the next time SFLUSH is used.

Any word which accesses a block or screen will prompt for it to be loaded from tape unless it is already in a buffer. Words such as INDEX and TRIAD which access several blocks in sequence will automatically load blocks after the first that they use. The same applies to screens containing --> (next screen) when they are compiled using LOAD.

The six buffers occupy the highest memory used by the language. Each buffer consists of a status cell (two bytes), 1024 bytes of data capable of holding one block and two bytes containing zero which are used as end of block markers if the block in the buffer contains source text. The status cell contains the number of the block in the buffer or zero if the buffer is unused. The signbit of the status cell is set if the block in the buffer has been updated. The word STATUS lists for each of the six buffers the address of the status cell, its contents and whether the block in the buffer has been updatd. This word is purely to aid the user of a tape system; the sequence n BLOCK should always be used to obtain the address of the first byte of a block irrespective of which buffer it happens to be in.

The tape virtual memory system is an emulation of the disc system referred to in published material on Forth. The operation of a disc system and some other details of virtual memory are given in the subsection entitled Virtual Memory System.

Using Forth

When Forth is loaded the message 'Z80 FIG FORTH 1.1' comes up on the screen together with the cursor. Forth is a naturally interactive language so it can be used immediately simply by typing instructions at the keyboard.

Type 4 5 + . then press return.

```
4 5 + .        9  OK
```

Each number or operator must be separated from others by at least one space but extra spaces may be used freely. This is a general rule in Forth. The result 9 should be printed on the screen followed by OK. Each token such as 5 and + is treated in the same way. First a list called the dictionary is searched to see if the token is the name of a word, which is the name given to operators in Forth. Then if the token is not a word it is tested to see if it is a number. If it is neither a word nor a number it is rejected. Words such as + and . are executed. Numbers such as 4 and 5 are placed on the stack which is a temporary store.

The example input line is processed in the following way. The number 4 does not appear in the dictionary but is a valid number, so it is placed on the stack. The same applies to 5 . + is a Forth word which adds the first two stack entries and replaced both with the result. The word . pronounced dot prints the first stack entry which is now 9. The message OK is then printed to show that the execution of the line is complete and that no errors have been detected.

Stack

When a number is entered it is placed on the stack, which is a temporary store arranged so that the last item placed in it is the first out when any item is removed. The next number entered can be visualised as pushing down the first and in its turn becoming the first stack entry. The number entered first is now the second stack entry. This organisation is sometimes described as last in first out.

```
2 3 4  OK
```

When the numbers 2 3 and 4 are entered they are put on the stack with 4 as the first stack entry because it was entered last. Each time dot is used one number is removed from the stack and printed. With very few exceptions Forth words remove their arguments from the stack when they are used. Other words are used to make copies of numbers on the stack. After the three numbers have been printed the stack is empty because there are no more items on it. If dot is used when the stack is empty then an error message results. Any operation or sequence of operations which causes the stack to empty will result in this error message.

Arithmetic

Arithmetic in Forth is carried out using reverse polish notation or RPN. This is not as baffling as it sounds. Instead of placing an arithmetic operator such as + between its operands (called infix notation) operators follow their operands. Some examples of postfix notation are given below, together with infix equivalents.

Postfix	Infix
2 3 +	2 + 3
2 3 + 5*	(2 + 3) * 5
3 5 * 2 +	2 + (3 * 5)

It should be apparent that the order of operations is always explicit in RPN. No brackets or precedence rules are ever used.

Forth can be used as a calculator simply by entering arithmetic in this postfix form. The calculations are carried out on the stack.

Keyboard entry	Stack contents
2	2
3	3 2
+	5
5	5 5
*	25

The word + adds the first two stack entries and replaces them with result. Similarly the word * (star) multiplies two numbers. The result of the calculation 25 is left on the stack until something is done with it explicitly. It could be used in another calculation or by another Forth word. In this case it can be printed using dot.

The use of a stack for arithmetic may well be new and unfamiliar. The best approach initially is to think in terms of just one number on the stack at any one time. Each individual arithmetic operation can then be entered in sequence.

100	OK	Start with 100. Enter it so that 100 is on the stack
12 +	OK	Add 12 to this number. 112 is on the stack.
14 /	OK	Divide by 14. The result 8 is on the stack
5 -	OK	Subtract 5 leaving 3.

The division word / (called slash in Forth) is an example of an operation which gives different results depending which way it is done. For example 112 14/ is 8 but 14 112 is 0.125. The order in which numbers are entered when non commutative words like / are used is quite natural.

Once postfix notation and the use of the stack become more familiar then wider use of the stack can be made. There are a number of words in Forth for this purpose.

Words and their use

Some words have already been met. + - * / and . are all words. Like nearly all words in Forth they can be used simply by typing them in.

Each word has a name made up of letters and other characters. By convention letters used in word names are always upper case letters. Word names are chosen to indicate what the word does. Many symbols used in English punctuation are used as the names of frequently used words and these include . , : ; ! and ?.

Each Forth system initially contains about 200 words. The user does not have to learn all these words to make effective use of the language but only those that are of interest or value to an application. Later it is explained how to add new words chosen and named by the operator. For the moment, though, only words already in the language can be used. If the name of a word not in the core language is used then it will be rejected followed by

? MSG 0 to show it has not been recognised.

Number bases

Forth can operate in any number base. For example the word HEX alters the base to hexadecimal or base 16. When this word has been used then numbers are entered in hex and results printed out in hex. The base can be changed at any time so numbers can be entered in one base and printed out in another allowing conversions.

```
HEX 9 7 +      10  OK      Nine and Seven add up to 10 in hex
13 DECIMAL .   19  OK      13 hex is 16 + 3 or 19 in decimal
```

HEX and DECIMAL are words which change the base in which Forth operates.

Forth is always in decimal initially. When bases higher than 10 are used the numbers after 9 are A B C .... using letters of the alphabet as necessary. When bases lower than 10 are used then only digits from zero to one less than the base are valid. In binary, for example, only 0 and 1 are numbers.

Bases other than decimal and hex are obtained by altering the value of the base variable BASE. A word! (pronounced store) is used to do this.

```
3  BASE !       OK        Stores 3 at BASE altering base to 3
101  BASE !     OK        Stores 101 base 3 (10 decimal) at
                          BASE
```

BASE is an example of a variable. The use of variables is described in more detail later in the section on constants and variables.

Core language words

The initial word set in Forth is sometimes referred to as the core language. These words fall into three categories. First the basic building blocks which are used to make up higher levels of the language and user defined words. There are about 40 of these and they perform such functions as adding two numbers and storing a number in the memory of the computer. Then there are a number of specialised words which interpret what is typed at the keyboard by the user. Finally there are some higher level words which perform more complex tasks for the user. Some of the words in the first and third groups are shown below.

Two words which control the display are SPACE and CR. These words output a space and a linefeed to the screen respectively. A word SPACES (separate from SPACE) takes one number from the stack and outputs that number of spaces. 8 SPACES outputs 8 spaces and so on. More generally a word EMIT takes a number from the stack and interprets it as an ASCII code value which is used

to print the appropriate ASCII character. HEX 41 EMIT prints the letter A.

There are several words which perform the usual arithmetical and logical functions on numbers. MINUS reverses the sign of a number so 27 MINUS . will print -27. ABS forms the absolute value of a number so 27 ABS. and -27 ABS. will both print 27. Logical functions include AND OR and XOR and these operate on a bit by bit basis so 2 1 OR . will print 3.

Numbers and other values can be compared using relational words. These words return one (true) if the relational test is satisfied and zero (false) if it is not. These truth values are called flags and can be used to make decisions as explained later in the section on conditional and loop constructs. Equality is tested by the word = (equal) so 5 5 =. will print 1 indicating true and 5 4 = . will print 0 indicating false. The words < and > make the tests less than and greater than respectively.

All the words in the language are listed in response to VLIST. The functions of all these words are described in the glossaries later in this guide.

Structure of Forth

A great deal has been said about the stack in Forth. Indeed there are two stacks used in the language; the parameter or data stack which has already been discussed and a second called the return stack. This latter stack normally operates without requiring the intervention of the user although its use is valuable for certain purposes.

The parameter or data stack (often just called the stack) is fundemental to Forth. It is the principal means of communication between the words and so between different parts of a Forth program. It is what makes Forth a modular and context free language. Context free means that each operation or word is executed in the same way whatever else is going on at the same time. Values or parameters needed by each word can be found on the stack and results are placed on the stack ready to be used by the next word.

The user is completely responsible for making use of and managing the stack. Many words require parameters to be on the stack when they are used and also leave the results of their operation on the stack . Whenever a word is formally described its effect on the stack is always specified. Numbers are put on the stack by typing them in or writing them as part of the text of a program. Nearly all words operate on only the first two or three numbers on the stack. Numbers below these are unaffected. A series of stack manipulation words exist to bring numbers into the required position near the top of the stack so they can be used. and to copy numbers so that they can be used more than once. Finally results of a series of operations are left on the stack

for printing out or passing to another program.

Although numbers have been referred to as the most common items on the stack a variety of different data types are dealt with on the stack. Each stack entry consists of two bytes together and since a byte is eight bits the stack is said to be 16 bits wide. Into these 16 bits can be put numbers, addresses, boolean flags, ASCII characters and any other type of data that will fit. Two stack entries can be used together to make 32 bits and this arrangement is used, for example, for double precision numbers. For other date types which would require several stack entries together such as a string it is more convenient to store the data in memory and manipulate its address on the stack.

For the mean time it can be assumed that all references to stack entries mean single entries of 16 bits. Most commonly these will be in the following categories:
integer numbers between -32768 and +32767
boolean flags where zero means false and any non zero value means true
characters where the lowest seven bits are the ASCII code of the character
addresses of memory locations.

Forth is often described as a typeless language. This means that whether a stack entry is an integer number of a character depends on the context in which it is found. It is entirely the responsibility of the user to keep track of the meaning of stack entries. This means both freedom from restrictions on what can and cannot be done with different types of data and the responsibility for avoiding errors by confusing different types of data.

In the next section the most commonly used stack manipulation words are introduced. These words are used to move values around the stack so that several words can be used together. Immediately after folows an introduction to defining new Forth words chosen by the user.

Stack manipulation words

These words are introduced here because they are among the most frequently used words in Forth. In addition to its use for arithmetic the stack is used extensively for passing data from one Forth word to another.

Nearly all Forth words remove their arguments from the stack when they are used. For example if . is used to print the number on the top of the stack then once the number has been printed it is no longer on the stack. In order to do more than one thing with a number a copy has to be made so that one operation can use the number and one can use the copy. This is accomplished by the word DUP which duplicates whatever number is on top of the stack. Using DUP followed by . the number on top of the stack can be

examined without altering it. DUP makes a copy and . prints the copy leaving the original number unaltered.

Below is shown the effect of DUP and some other words on a stack on which the numbers 5 4 3 2 1 have been placed by typing them in. The number 1 is on top of the stack because it was typed last; 2 is the second stack entry and so on.

Word	Stack contents before	Stack contents after
DUP	1 2 3 4 5	1 1 2 3 4 5
SWAP	1 2 3 4 5	2 1 3 4 5
OVER	1 2 3 4 5	2 1 2 3 4 5
2DUP	1 2 3 4 5	1 2 1 2 3 4 5
DROP	1 2 3 4 5	2 3 4 5

SWAP reverses the order of the first two numbers on the stack so that the second number can be used easily. When a copy of the second number is needed then OVER provides a more convenient solution that SWAP DUP because the existing first and second entries are not moved. 2DUP copies the first and second entries allowing a comparison for example and is the exact equivalent of OVER OVER. Finally DROP is used to discard unwanted stack entries. It is important that this is done otherwise the stack will build up indefinitely and eventually overwrite the language.

These words can easily be used to make arithmetic operations more comprehensive. For example the square of a number can be produced by DUP * which multiplies the number by itself.

Defining new words

The cube of a number can be calculated in the folowing way.

Keyboard entry	Stack contents
4	4
DUP DUP	4 4 4
*	16 4
*	64
.	64 OK

The sequence DUP DUP * * takes the first stack entry and returns the cube of that number. So a new word CUBE can be made which will calculate the cube of any number. The number is taken from the stack and the cube of the number is returned to the stack. The word: (colon) is used to form the new word CUBE and colon is called a defining word.

```
: CUBE DUP DUP * * ;  OK
```

This line, which should by typed in as it is written, is referred to as a definition. The definition starts with colon. After : comes the name of the new word CUBE, which may be formed from any letters, numbers and other ASCII characters except a space. If a number is used alone though its role as a word name will effectively prevent its use as a number so at least one non numeric character is usually included.

The words DUP DUP * * are not exectued in the usual way. Instead they are compiled into the definition of the new word CUBE. Another word semi, which is entered as ; is used to end the definition. The new word CUBE is then complete and the language returns to the interactive mode which means that other keywords typed in will be executed and not compiled into CUBE. The message OK shows that CUBE has been accepted and that the definition did not contain any errors.

The word CUBE is now ready for use.

```
3  CUBE .       27   OK
7  CUBE .       343  OK
```

This new word behaves exactly as DUP DUP * * do when typed in successively.

Number literals

When a number appears in a colon definition it is compiled into the new word and is called a literal. When the word containing the number literal is executed the number is pushed to the stack.

```
: 2CUBE CUBE 2 * ;     OK
3 2CUBE .         54   OK
```

This example uses the previously defined word CUBE. Building up definitions in this way is the essence of using Forth. When 2CUBE is executed CUBE leaves the cube as the first stack entry. The number 2 is pushed to the stack and * multiplies the two numbers at the top of the stack, leaving the result on the stack.

String literals

Strings or messages can be made part of a word. When the word is used the string is printed on the screen.

```
: CUBE. CUBE CR ." Cube is " SPACE . ;   OK
5 CUBE.
Cube is 125    OK
```

Here the new word is called CUBE. . The suffix . suggests it prints its answer. CUBE is used again to calculate the cube and leaves it on the stack. CR feeds a line. The word ." compiles the string which follows it until the closing quote " into the word. ." is pronounced dot quote. SPACE leaves a space and

finally . prints the result

A string literal may be the only element of a word which is then used in other word definitions.

```
: A$   ." A frequently used string " ;  OK
A$          A frequently used string     OK
```

There are many more flexible ways of using strings in Forth and some are discussed later on.

New words and the stack

The original definition of CUBE could have been designed to print its result immediately by including . in the definition. This would have been the easiest thing to do if the calculation of a few cubes was all that the user required. If a word returns its result to the stack then it becomes a much more flexible instrument. It can be used to convert the first stack entry to its cube in any context. Other numbers can be on the stack as second and subsequent entries and they will not be affected by CUBE. Once the user has defined CUBE he no longer has to worry over the details of calculating cubes nor will he make errors over them. These principles become important when several words are used together to make more powerful words until one word is the program that meets the user's application.

Three other points are helpful when defining words. Words should be given meaningful and descriptive names. Definitions should be kept short and simple so that complex functions are formed from several words rather than one and each word 'hides' a distinct operation. Finally words can be tested by putting dummy arguments on the stack and checking the operation of the word, so bringing to light errors and limitations. Below is the program for testing CUBE.

```
0    CUBE  .        0    OK  Zero gives correct result
-2   CUBE  .       -8    OK  Negative numbers give correct result
32   CUBE  .   -32768    OK  Overflow in single precision
                             arithmetic (use double precision
                             for numbers higher than 31)
```

Errors in definitions

A correct definition is indicated by the message OK or the message IS NOT UNIQUE (or MSG # 4). This latter message is not an error message in the usual sense but simply shows that the name chosen for the new word is also the name of an existing word. This is perfectly permissible and any name can be reused, including those of words in the core language. A consequence of reusing a name though, is that when that name is used only the most recent name will effectively be lost. Words using these earlier words will not be affected though. There is a means of

using the same word name more than once for different words and making all the different versions accessible. This is called vocabulary control and is explained later.

Only words already in the dictionary and numbers and strings can be used in definitions. A missed space will make a token unrecognisable and result in such error messages as:CUBE ? . Stack underflow during a definition will be detected as an error. If these or other errors occur during compilation of a new word then the partially completed word will be erased.

Discarding unwanted words

An unwanted word is removed using FORGET followed by the name of the word to be removed.

```
FORGET CUBE        OK
```

FORGET causes the removal of the word that is forgotten and all subsequent entries. However if there is more than one word with the name which follows FORGET then only the most recently entered version is forgotton. In normal use attempts to forget core language words are rejected although this restriction can be removed if required as is usually the case in Forth. All words outside the core language are forgotten when COLD is used.

Conditional and loop constructs

Forth has a rich set of conditional and looping constructs. These are available only within compiled definitions of words.

As with most languages the boolean values true and false which govern the operation of these constructs are defined as follows:

false	zero
true	any non zero value

Any number on the stack can be used as a truth value and is referred to as a flag on the stack. There are a number of words which perform relational tests such as comparing two numbers to see which is larger or seeing if a number is equal to zero. These words leave a flag on the stack according to the result of the test they have made. This flag can then be used by such conditional words as IF and WHILE.

IF ENDIF

The simplest conditional construct in Forth is IF ENDIF.

```
:  UNITY 1 = IF ." Unity " ENDIF ;   OK
1  UNITY     Unity    OK
7  UNITY              OK
```

This example can be compared with the same program in a Pascal like language.

```
PROCEDURE UNITY
IF X = 1 THEN (
                WRITE "Unity")
ENDPROC
```

In Forth conditionals are expressed in a postfix form so the words associated with a conditional operator such as IF or ENDIF precede the operator instead of folowing it. There is no need for brackets to indicate the scope of the statements to be executed if the conditional test is satisfied because ENDIF shows the end of these statements.

The word = (equals) is a relational operator which generates a true flag if the two entries at the top of the stack are equal and a false one if they are not. Here the first stack entry when UNITY is used is compared to 1. The word IF takes the flag formed by = from the stack. If this flag is a true non zero value then the words between IF and ENDIF are executed; otherwise they are not. Execution resumes unconditionally after ENDIF.

Every IF construct must be completed by an ENDIF in the same word definition. A word ELSE can be used to specify an alternative set of words to be executed if the test made by IF is false.

```
:  UNITY2 1 = IF    ." Unity"
              ELSE  ." Not unity"
              ENDIF ;  OK
```

If the flag on the stack examined by IF is true then only the words between IF and ELSE are executed. If the flag is false then only the words between ELSE and ENDIF are executed. In either case execution resumes unconditionally after ENDIF.

The word THEN is provided and may be used instead of the word ENDIF wherever it occurs if the user prefers.

DO LOOP

The finite loop structure DO LOOP is a simple but very valuable structure. It takes two arguments from the stack, both integer numerical values and not flags.

```
:  IOTA 10 0 DO I . LOOP ;   OK
IOTA        0123456789    OK
```

The example word IOTA contains the major elements of this construct. The word DO takes two entries from the stack. The first stack entry (entered second) is the counting index of the loop, usually just called the index. The second is referred to as the limit. On each passage through the loop the index is incremented by one and compared with the limit. As soon as the index equals the limit the loop is no longer executed. Inside

the loop the word I places a copy of the index on the stack where it can be used.

Each DO construct must be completed by a LOOP in the same word definition. The index and limit may be any values. The limit is written first because it is more often passed as an argument.

```
:  CUBES 0 DO CR I DUP . CUBE . LOOP ;   OK
4  CUBES
0 0
1 1
2 8
3 27     OK
```

CUBES allows the generation of a cube table of any size. The CR feeds a line after each number and its cube. Two copies of the loop index I are made using I DUP. The first is printed and the second cubed and printed.

BEGIN UNTIL

This structure is a loop whose execution is controlled by a flag on the stack.

```
:  COUNT 4 BEGIN DUP . 1+ DUP 9 = UNTIL DROP ;   OK
COUNT        4 5 6 7 8     OK
```

The BEGIN UNTIL loop is always executed at least once. At the bottom of the loop UNTIL takes a flag from the stack. If this flag is true the loop terminates; if it is false the loop is repeated until the flag is true. COUNT starts with a number on the stack. This number is printed, incremented and compared with 9. The loop continues until the expression DUP 9 = gives a true result. The left over number is discarded by DROP which is used to remove an unwanted number from the stack. Unwanted numbers should always be removed or they will cause errors later in a program.

BEGIN WHILE REPEAT

This loop complements the BEGIN UNTIL loop by providing a looping structure in which the conditional test is made at the start of the loop.

```
:  COUNT2 4 BEGIN DUP 9 < WHILE DUP . 1+ REPEAT DROP ;  OK
COUNT2              4 5 6 7 8   OK
```

The BEGIN WHILE REPEAT loop is a WHILE construct expressed in postfix form. The conditional expression governing the execution of the loop precedes WHILE and form a flag which WHILE takes from the stack. If this flag is true then the instructions up to REPEAT are executed, a loop back to BEGIN occurs and the conditional test between BEGIN and WHILE is repeated. Execution of the loop continues until WHILE takes a false flag from the

stack. The loop may not be executed at all if the test is not satisfied the first time it is made.

Constants and Variables

Constants and variables are used in Forth to enable a value to be referred to by name. Fewer constants and variables are used compared with other languages for a similar application though, as a general rule because Forth users have the stack and number literals at their disposal.

A number literal can be used in only one word so if the same value is to be used by several words then it should be made a constant. A constant should also be used where giving a name to a value makes a program easier to understand. As its name suggests a constant should usually be used for numbers which do not alter in value during a program. It is possible, none the less, occasionally to alter the value of a constant.

During the execution of a program many numerical and other values are generated which are temporary. The stack allows these values to be dealt with in Forth whereas in other languages variables (or arrays) have to be created for each of them. Values which are fundamental to a program or which are used at different times should be assigned as variables. When this should be done is a matter of judgement. A word which references a variable cannot be used in isolation from the variable and so loses modularity. Against this use of variables makes programs easier to understand and less prone to error.

Defining constants and variables

Constants and variables are kept in the dictionary and have names like other words. Instead of being defined using colon though, they have defining words of their own which are used a little differently from colon.

```
12  CONSTANT DOZEN    OK
57  VARIABLE HEINZ    OK
```

A constant is defined with its value which is 12 in this example. A variable is supplied with its initial value (57). These values may be zero if no particular value is needed as an initial value but failure to specify some value will result in an error. CONSTANT and VARIABLE are defining words like colon but their action is limited to the number which precedes them and the name which follows them and they do not cause other words to be compiled and not executed in the way that colon does. Immediately after the defining word comes the name of the new constant or variable.

Using constants and variables

Using constants is easy and involves no new ideas.

```
DOZEN .                 12  OK
: BAKERS DOZEN 1+ ;         OK
```

When the name DOZEN is used the value of the constant is pushed to the stack. It can be printed using . in the usual way. A constant can be incorporated into another word using its name. The word 1+ in BAKERS adds one to the value left on the stack by DOZEN and thus leaves 13 on the stack.

Variables operate a little differently from constants. A variable can be thought of as a place where a value is kept and so a variable pushes the address and not the value of the variable. In order to recover the value of the variable the word @ (fetch) is used. This word replaces the address with the value stored at that address. Another word ! (store) is used to place a new value in the variable.

```
HEINZ @ .                   57  OK
62 HEINZ !                      OK
HEINZ @ .                   62  OK
```

The user does not need to enter the address at any time; it is always referred to by the name of the variable. The use of @ and ! may appear tedious in isolation. They serve to make variables more flexible in complex applications. The word store takes the place of the assignment operator in other languages. Instead of writing HEINZ = 62 or HEINZ := 62 Forth uses 62 HEINZ ! .

```
:  SOUP 41 HEINZ ! ;            OK
:  FISH 15 HEINZ ! ;            OK
:  MENU HEINZ @ . ;             OK
FISH MENU                   15 OK
```

If a large and apparently meaningless number is encountered when using a variable it is probably the address of the variable and indicates a forgotten @ . The address itself is useful in some applications and is obtained simply by using the name of the variable without @. The words @ and ! can be used for any memory location and not just variables and are extremely useful operators. There is a word ? which is defined as @ . and prints the value of any memory location. It can be used with variables.

```
HEINZ ?                     57 OK
```

Screen Editor

The editor is made up of Forth words which are used to create and edit text. The text may consist of word definitions and other instructions and is stored on screens of 1024 characters arranged in 16 lines of 64 characters each. The screens are stored in the blocks which the mass storage is divided into and referred to by

number. Screens 4 and 5 are usually reserved for error messages. The relevant section entitled OPERATION should be consulted for hardware and operating system dependent details of disc drives and other mass storage devices.

The MSX editor resides in the EDITOR vocabulary (see later for details on vocabularies) and makes use of the screen editing facilities provided in the MSX ROM.

In order to edit a screen, type the word n EDIT where n is the screen number. If the screen is already resident in a buffer then this will cause the screen to be listed and the edit mode entered. If not then the system will load it from the tape first.

In the case of a new screen it is advisable to use the word n CLEAR first before using the word EDIT to ensure that the screen is properly initialised. This will also cause the screen to be loaded if necessary. (As a short cut it is possible to trick the system into thinking that it haas already loaded a screen by pressing CONTROL-STOP after the reading prompt. This will cause error # 8 to occur and the reading operation to the aborted. STATUS will reveal that the system thinks it has loaded the screen. Typing in n CLEAR again will ensure that the screen is initialised and you are ready for editing using EDIT. This short cut can be useful if you wish to use a screen for a temporary program that you dont want to save and you dont have a tape to hand!)

After listing, the system is in EDIT mode and the cursor controls may be used to move around the screen and write or modify text on any of the lines.

Each line is preceded with a line number. There must be at least one space between the number and the text. When a line has been typed it must be entered into the machine using the return key just like in BASIC. Be careful to remember this as it is easy to think that the data is there because you can see it on the screen!

As just stated, lines consist of 64 characters. Unfortunately, the MSX has a 40 line display. This means that potentially each FORTH line can occupy one and a half lines on the screen. During typing if you wish to extend over the end of a line then the following scheme should be used to overcome the peculiatrities of the MSX editor. Just prior to the end of the line go into insert mode and insert some spaces or text . This will cause the current line to be extended to two lines by scrolling the lines below down one position. You may then continue typing the line as normal. Note that only 64 characters will be accepted when you press return. Any in excess of this will be ignored.

When you have finished editing press CONTROL-STOP this will cause you to exit the editing mode and the OK prompt will return. For neatness it is advisible to scroll to the bottom of the screen before doing this. If any changes have been made to the text

then the screen will be marked as UPDATED. This may be checked using STATUS if desired. Remember to use FLUSH or SFLUSH to record your changes on the tape before switching the machine off or executing any untried program that may crash the machine!

When a screen is getting full, it will not be possible to display the whole of the screen in one go. To allow editing of the parts that scroll off the top when the EDIT command is used, the user may press any key while the screen is listing. This will cause the rest of the listing to be aborted and the edit mode to be entered as normal.

There is nothing special about the line numbers on the screen. They are simply there as a prompt to the user. This can be used to advantage during editing. If you wish to move or reorder lines on the screen simply type over the current line number with a new one. Remember to press return afterwards to record the change. The only limitations are that the line number must be between 0 & 15 and there must be a space bewtween the number and the text. Quiting and re-listing the screen with EDIT allows you to extensively modify screens without getting confused.

One point should be noted whilst writing programs. FORTH treats text screens as a continuous block of 1024 characters. The separate lines are purely for users convenience. This means that there is no inferred space at the end of one line and the beginning of the next. If a FORTH word uses the last character position of a line be sure to leave a space at the start of the next to ensure that the interpreter doesnt run two words together and cause an error.

Screens may be moved around using SCOPY. The sequence 50 55 SCOPY copies screen 50 to 55.

There is a direct mapping between screens and the blocks of the virtual memory system. For the moment the two may be regarded as synonymous. There is no screen or block numbered zero. Further details of the relationship between the virtual memory system and text screens are given in the subsection of Forth Techniques entitled Virtual Memory System.

Using Text Screens

The purpose of text screens is to allow colon variable constant and other definitions and instructions to be stored in permanent form. Several of these may make up a complete program covering several screens.

In general any text which makes up a valid executable input line can be stored on a screen. When the screen is compiled (called loading in Forth) the text on it is treated in exactly the same way as input lines are treated. A screen is loaded by typing n LOAD where n is the screen number. If the text on a screen contains an error it will be reported in the same way that an

error in a input line is reported. In addition the editor word WHERE will show its position on the screen.

Any of the example word definitions given earlier in this section, such as CUBE, may be edited on to a screen and then loaded. Select an unused screen such as screen 50. Using the editor, first clear it with CLEAR (this is important), then edit the definition on to it. Next load the screen by typing 50 LOAD. If any error occurs (except MSG 4) correct the text using the editor. It will be recalled that error MSG 4 simply indicates that a word name has been used more than once which is unimportant in this context.

A comment may be placed on a screen by enclosing it in round brackets (and). The usual space must be left between the brackets and the comment text. Text within the brackets will be ignored when the screen is loaded.

When related text covers more than one screen, screens may be linked by --> pronounced next screen. If this word is placed on screen 50, at the end of the text on that screen, then the sequence 50 LOAD will also load screen 51. This may be extended indefinitely over consecutive screens. One screen may also load another by a LOAD instruction. For example 52 LOAD could be written on screen 50. One screen may contain instructions to load several others and is then referred to as a load screen. This sort of load instruction should not be chained through several screens though.

Interpretation of the text on a screen is ended by ;S or in any case at the end of the screen.

A title in the form of a comment is usually placed on line 0 of each screen and this allows screens to be searched using INDEX. For example 1 5 INDEX lists the first lines of screens 1 to 5.

Forth Techniques

This section contains more detailed explanations of some of the techniques used in Forth programming.

Parameter and Return Stacks

The parameter stack is the stack explicitly available to the user for the temporary storage of data and passing parameters between words. It grows downwards in memory from its base, whose address is stored in the user variable S0 (szero). The parameter stack shares an area of memory with the dictionary, the dictionary growing upwards from low memory and the parameter stack growing downwards from high memory. Usually several thousand bytes separate the two so there is room for the parameter stack to grow to any practicable depth.

Generally only a few items should be manipulated on the parameter stack at any one time so that the parameter lists for individual words can be kept simple and easy to use. Each word should leave only defined parameters on the stack; other values should be discarded or they will clutter the stack and make errors difficult to avoid. Two types of error are likely to occur when using the stack. If the stack grows so large it encroaches on the dictionary it is said to have overflowed whereas if the stack is empty and an attempt is made to remove a value from it then it is said to have underflowed. The latter error is much more common and is very easily made. The stack is tested for overflow and underflow after each word is executed but not during the execution of a word. This makes execution fast but places on the user the responsibility of ensuring that appropriate parameters are on the stack for each word to use.

The value of the stack pointer, the address of the first stack entry is placed on the stack by SP@ and the sequence SP@ . will print the value of the stack pointer while SP! clears the stack.

There is a second stack in Forth called the return stack. This stack, which is used by the language for the storage of return addresses generated when one word calls another, can also be used for the temporary storage of values. There are some restrictions on this use but it adds a valuable degree of flexibility to the parameter stack.

For example if there are two items on the parameter stack and the second entry is to be used for some purpose the word SWAP is often used to make it accessible. Another way is to use >R (to R) to move the first parameter stack entry to the return stack leaving what was the second parameter stack entry on top. The value now on the return stack can also be accessed by the word R which copies it to the parameter stack without altering the return stack.

R can be used any number of times in a word so the return stack value can conveniently be used several times in a loop for example. The value is recovered by R> which moves one value from the return stack to the parameter stack. A number of values can be shuffled between the two stacks in this way.

The principle restriction is the >R and R> may only be used in a word definition and each >R must be balanced by and R>. No net changes to the return stack may be made by a word and the return stack cannot be used to pass values betwen words. Additional care is needed when using these words inside a DO LOOP construct because the index and limit are stored on the return stack. The word R which merely copies the first return stack entry to the parameter stack without altering the return stack may be used freely.

The return stack grows downwards in memory from its base which is stored in the user variable RO (rzero). The value of the return stack pointer is placed on the parameter stack by RP@.

Relational and Logical Operators

These operators are used to make comparisons and tests, returning flags which are most often used by the conditional and looping structures. There are a number of words which compare two single precision numbers on the stack. All of these except U< assume the numbers being compared to be signed. U< is used to compare unsigned numbers, especially memory addresses. There are two unary relational operators, 0= and 0<. 0= can be used as a logical NOT to reverse the state of a flag and 0< to test the sign bit of a number.

The flags from separate tests can be combined using the logical operators AND, OR and XOR. For example the sequence XLEN @ 1 < XLEN @ 16 > OR where XLEN is a variable will generate a flag which will be true if the value of XLEN is either less than one or greater than sixteen. More complex conditional statements can be written in this way.

A little care sometimes has to be taken with logical operators. The flags returned explicitly by relational operators always use one to denote a true result. Combining two or more of these flags with logical operators such as AND will give the expected result. But if the flag is generated by an implicit test such as - for inequality between two numbers then a (perfectly valid) true result might be expressed as a non zero value other than one. Since the logical operators work on a bit by bit basis this may give an erroneous result. For example 1 1 AND is 1 but 1 2 AND is 0. If a logical operator is being used then a test such as 41 - should be written as 41 = 0=.

Conditional and Looping Constructs

This sub section includes illustrations of the applications of the conditional and looping constructs to dumping the contents of memory in hex format.

```
HEX

: BYTES         DO I C@ 2 .R SPACE LOOP ;
: LINE          CR DUP 0 5 D.R SPACE
                DUP 8 + SWAP BYTES ;
: DUMP          HEX DUP 80 + SWAP
                DO I LINE ?TERMINAL
                IF LEAVE ENDIF 8 +LOOP ;
: DUMP2         HEX BEGIN DUP LINE 8 +
                ?TERMINAL UNTIL DROP ;
```

The first word BYTES illustrated a DO LOOP construct where both the index and limit are supplied from outside the word via the stack. The word .R uses two stack entries printing the second as a signed number right adjusted in a field whose width is the first stack entry. Thus I C@ 2 .R SPACE fetches the byte whose

address is the value of the index I and prints it suitable formatted. BYTES can be tested by a sequence such as 8 0 BYTES which would dump the first eight bytes of memory.

LINE dumps eight bytes on a line starting with an address taken from the stack. The address itself is printed at the start of the line in a field of five spaces followed by a space. A zero is pushed in front of a copy of the address on the stack so that all addresses, including those above 8000H, can be printed as positive double precision numbers. The word D.R outputs a signed double precision number right adjusted in a field of spaces whose width is the first stack entry. DUP 8 + SWAP form an index (the address) and a limit (the address plus eight) on the stack for BYTES.

DUMP uses the DO +LOOP construct. This behaves similarly to the DO LOOP construct except that on each passage through the loop +LOOP takes a value from the stack which is used to increment the loop index whereas LOOP does not take a value from the stack and always increments the loop index by one. The value taken from the stack by +LOOP may be either positive or negative. DUMP takes an address from the stack and dumps 128 bytes. After each line of eight bytes the index of the loop, used by LINE, is incremented by eight.

The word HEX at the start of the definition ensures that the output is always expressed in hex although the starting address may be specified in any base. DUMP also shows the use of the word LEAVE which terminates a loop the next time LOOP or +LOOP is reached, irrespective of the number of times the loop has been executed.

The word ?TERMINAL returns a true flag only if a key has been pressed. This flag is tested by IF and LEAVE is executed only if the flag is true. When DUMP is used pressing any key causes the dump to stop at the end of the current line. In this example an IF ENDIF clause is nested inside a DO LOOP clause. The conditional constructs may all be nested in this way but the scope of one construct must be wholly contained by the construct within which it is nested. For example DO DO LOOP LOOP and IF DO LOOP ENDIF are valid sequences but IF DO ENDIF LOOP is not. Several levels of nesting may be used although it is best to use more and shorter word definitions and fewer levels of nesting within a word definition wherever possible.

DUMP2 shows how LINE can be placed inside a BEGIN UNTIL loop. On each passage through the loop LINE uses a copy of the current value of the address and the address is incremented by eight. The flag generated by ?TERMINAL is tested by UNTIL and the loop is executed until this flag is true. Memory is dumped indefinitely until a key is pressed.

The second set of examples shown here is concerned with dumping memory in ASCII format. The main problem to solve is the treatment of control and other non printing characters.

```
HEX

: ASCBYTE           DUP 20 < OVER 7F = OR
                    IF  DROP 2E EMIT
                    ELSE EMIT
                    ENDIF SPACE ;
: ABYTES            DO SPACE I C@ ASCBYTE LOOP ;
: ALINE             CR DUP 0 5 D.R SPACE
                    DUP 8 + SWAP ABYTES ;
: ADUMP             DUP 80 + SWAP
                    DO I ALINE ?TERMINAL
                    IF LEAVE ENDIF 8 +LOOP ;
```

The role of ASCBYTE is to take and ASCII character from the stack and print it if possible or else to print a dot. A copy of the character is tested to see if it has a value of less than 20 hex and a second copy to see if it has a value of 7F hex. The flags resulting from these two tests are combined with the logical word OR and if either of them gives a true result then the unprintable ASCII character is dropped and a dot (2E hex) is printed instead. The three words ABYTES ALINE and ADUMP follow the pattern of BYTES LINE and DUMP above.

The appearance of the output generated by these dump words could be improved by using the number formatting words to produce number output right adjusted in a field of zeros. They could also be more tightly, though less legibly, written but this is not usually desirable unless time is a critical factor.

The DUMP words shown above are already included in the core language supplied.

Numbers and Arithmetic

The core language contains operators for manipulating single and double precision integers. Internally these are represented in binary twos complement form with single precision numbers occupying one stack entry or two bytes and double precision numbers occupying two stack entries or four bytes. Numbers can be input and output in any base according to the value of the user variable BASE.

When a number is entered it is converted to binary and pushed to the stack. An ordinary number which is a sequence of digits, possibly preceded by a minus sign, is automatically treated as single precision. The two bytes used by these numbers allow the representation of 65536 distinct integer values. This range is used in two ways:

```
0 ...          32767    32768 ...    65535
0 ...          32767   -32768 ...       -1
```

Either positive numbers from 0 to 65535 or both positive and negative numbers from -32768 to +32767 can be represented. The same binary values are used in each case and whether they represent unsigned or signed numbers depends on the context. The operator . (dot) which prints the first stack entry always interprets a number as signed. There is another operator U. (udot) which prints the first stack entry as an unsigned number.

```
-1              .          -1      OK
-1              U.      65535      OK
```

Unsigned numbers are particularly relevant to memory addresses since the range (0 to FFFF in hex) alows any part of memory to be accessed.

The words + - 1+ 2+ can be used with both signed and unsigned numbers. All the other single prcision operators treat numbers on the stack as signed.

Double precision numbers have a range of -2 147 483 648 to 2 147 483 647 signed and 0 to 4 294 967 295 unsigned. A double precision number is entered by including a point as part of the number. The position of the point may be meaningful to the user but is otherwise unimportant. The number needs two stack entries with the most significant part nearer the top of the stack. It can be printed using the operator D. (ddot).

```
12              .          12      OK
12.             D.         12      OK
12.             . .        0 12    OK
```

The last example shows that of the two stack entries which make up the double precision number 12 the first is zero because 12 is a positive number within single precision range.

Since Forth is a typeless language single and double precision numbers are not formally separated. Both types can be used in the solution of a problem. The word S->D converts signed numbers from single to double precision; the reverse is effected simply by discarding the high order part. Conversion is often avoided by using the mixed precision operators.

An unusual operator is */ (star slash) which is used to express fractional quantities.

```
: RATIO   2 3 */ ;
2712 RATIO .   1808   OK
```

The number 2712 is first multiplied by 2 and then divided by the first of the three stack entries 3. The intermediate value is double precision. The value of PI can be expressed to six figures as 355 113 */.

Another useful word is /MOD which gives the exact result of a division by returning both the quotient and remainder. For example seconds can be converted to hours minutes and seconds by two successive uses of the phrase 60 /MOD. The words MAX and MIN are used to restrict the range of numbers. MAX takes the first two stack entries and returns the larger; MIN leaves the smaller. The sign of a number is reversed by MINUS and its absolute value given by ABS. There are double precision equivalents of these words DMINUS and DABS.

Twos complement numbers may be new to some users. Briefly this means that negative numbers are formed by subtracting their absolute value from zero. Overflow is always ignored so the next number after -1 which is also 65535 as an unsigned single precision number is zero. Similarly if two positive single precision numbers are added to give a sum greater than 32767 there is overflow and a negative result. But the overflow is not reported as an error because the result interpreted as an unsigned number is correct.

High level defining words

The words : (colon) CONSTANT and VARIABLE are all used to define other words. They are referred to collectively as defining words. They perform the compiling function in Forth. In Forth the defining words and so the compiling process can be modified and added to just like the rest of the language. This section examines how this is done for words which handle data like CONSTANT and VARIABLE. The compilation process started by : is usually concerned with sections rather than data and is considered separately.

Since variables are a basic idea found in nearly all languages they tend to be taken for granted. In Forth variables are a convenient starting point for considering how to manipulate data. A variable is simply a place to put a value. The variable is given a name so it can be used conveniently. The name refers to the place and the value in the variable can be used or altered by getting it from the place where it is kept. This process is explicit in Forth. Using the name of a variable places the address where the value of the variable is kept on the stack. In order to use the value of the variable the word @ (fetch) has to be used to replace the address with the value on the stack.

New variables are defined using the defining word VARIABLE in a sequence such as 57 VARIABLE HEINZ. The action of the defining word VARIABLE is to make a new entry in the dictionary for the word HEINZ, reserve a place which is where the value of the variable will be kept and put its initial value of 57 there and then ensure that when HEINZ is used it places on the stack the address in memory where the value of the variable is kept. All these actions are controlled by the definition of the defining word VARIABLE. Below is shown a possible definition for VARIABLE.

```
: VARIABLE <BUILDS , DOES> ;  OK
```

The three elements of this definition <BUILDS , (comma) and DOES> correspond to the three actions of VARIABLE described above. The first word <BUILDS ensures that VARIABLE makes an entry in the dictionary . The entry contains the name of the new variable such as HEINZ and some other information and is often referred to as a header. Its format is common to nearly all words in the dictionary.

Secondly , (comma) takes the initial value of the variable 57 which is on the stack and puts it next to the header. This is the place where the value of the variable is kept.

Finally DOES> ensures that VARIABLE ensures that when HEINZ is used it leaves on the stack the address of the place where the value of the variable HEINZ is kept.

Three distinct actions are involved when <BUILDS and DOES> are used. First a defining word such as VARIABLE is defined using <BUILDS and DOES>. Secondly a particular variable such as HEINZ, which is one of many such variables is defined using the defining word VARIABLE. Thirdly the new variable HEINZ is used leaving its address on the stack.

The words <BUILDS and DOES> are called high level defining words. These are most important words which allow the creation and manipulation of data structures chosen by the user as single units. Each type of data structure has a defining word. The defining word uses <BUILDS to specify how the data structure is built up and DOES> to specify how the data is accessed. They form one of the most powerful features of the language.

Examples of data structures include variables, constants, arrays and string variables. Variables and constants are predefined in the core language. Forth offers the user freedom to add new data structures without any limit on their complexity or degree of specialisation.

The high level defining words are used in the following way.

```
: defining word name <BUILDS defining time code DOES> run time
code;
```

When the defining word is used <BUILDS creates a dictionary header for the new word. The code folowing <BUILDS specifies the form of the new word in the dictionary and inserts the initial values of the data. DOES> leaves the address of the start of the data on the stack when the new word is used. The code following DOES> is executed when the new word is used and usually accesses the data using the address of the first byte supplied by DOES>. These actions can be illustrated by the definition of CONSTANT shown below.

```
: CONSTANT  <BUILDS , DOES> @ ;
```

The defining word CONSTANT is used to define constants in a sequence such as 12 CONSTANT DOZEN. The code following <BUILDS in the definition of CONSTANT is , (comma) which ensures that CONSTANT places 12, which is the initial value of the constant DOZEN, in the parameter field of DOZEN. The code following DOES> is @ (fetch). The code is executed when DOZEN is used and uses the address left by DOES> to recover the value of the constant. The inclusion of @ accounts for the difference in the behaviour of constants and variables. Constants return their values and variables the address of their values.

The next example shows the definition of a defining word ARRAY which is used to define one dimensional arrays. ARRAY is used in the following way :

50 ARRAY AVERAGES	Defines an array AVERAGES dimension one by fifty. The initial value of each element of the array is zero.
41 AVERAGES	Leaves on the stack the address of the 41st element of the array AVERAGES. Each element can be manipulated using @ ! and +! in the same way as a variable.

The definition of the defining word ARRAY is shown below.

```
: ARRAY         <BUILDS DUP ,
                        DUP 2 * HERE SWAP ERASE
                        2 * ALLOT
                DOES>   SWAP 1 MAX OVER @ MIN
                        2 * +                   ;
```

The words after <BUILDS allocate space in the dictionary for the elements of the array and fill this space with zeros thus initialising each element to zero. Two bytes are needed for each element and two bytes at the beginning store the size of the array.

DUP	Writes a copy of the array size in the parameter field of the array (50 in AVERAGES)
DUP 2 * HERE SWAP ERASE	Erases or fills with zeros two bytes for each element from the dictionary pointer given by HERE.
2 * ALLOT	Reserves two bytes in the dictionary for each array element by advancing the dictionary pointer.

The data structure is now a size which is used for checking the validity of subscripts followed by the array itself.

When an element of the array is called by a sequence such as 41 AVERAGES, DOES> and the words after it come into play. These words check the validity of the subscript and return the address of the desired element.

Word	Stack contents	first entry at	left
DOES>	data address	subscript	The subscript is supplied when the array is used.
SWAP	subscript	data address	
1 MAX	subscript	data address	Minimum subscript value of one.
OVER @ MIN	subscript	data address	Maximum subscript value of size. Size is stored at the data address.
2 * +	element address		Left on stack.

The action of DOES> is to leave on the stack the address of the first byte of data referred to here as the data address. All other data is accessed by adding an offset to this address. In the case of arrays defined using ARRAY the first two bytes contain the size of the array. The address of the first subscripted element is the data address supplied by DOES> + 2. The subscripts are from 1 to 50 in the case of AVERAGES and the address of each element is found by adding two times the subscript value (there are two bytes for each element) to the data address. If the subscript supplied via the stack to AVERAGES is less than one then the address of the first element is returned; if the subscript is greater than 50 then the address of the last element is returned. This arrangement is sufficient to prevent corruption of memory outside the array. Instead an error message such as ARRAY SUBSCRIPT OUT OF BOUNDS could be generated or if speed is a priority then no error checking at all may be appropriate. The user is free to choose.

Once ARRAY has been defined then many different arrays of varying sizes can be added using ARRAY. Each array will behave in the same way. Each array will consist only of a dictionary header (so it can be found in the dictionary) and passive data. The active instructions are in the defining word ARRAY after DOES> ; this set of instructions is shared by all the arrays defined by ARRAY which is a highly efficient arrangement.

Vocabulary Control

The dictionary of the core language is a single linked list of word names. Each word name contains a pointer to the previous entry in the dictionary. The text interpreter begins a search for a word name at the latest dictionary entry. If a new word is defined using a name which already exists then the original definition is no longer accessible because the new one is found first and therefore used. Other words which contain the older version as part of their definition are not affected because their compiled code contains the address of the older version.

Still it might none the less be useful for one or more words to share the same name and yet be accessible and this is achieved by vocabulary control.

Vocabulary control provides the primary means of grouping together in the dictionary words with related functions. The core language contains a vocabulary word FORTH. A vocabulary word has in its parameter field a pointer to the latest entry in the vocabulary. Each time words are added to the vocabulary this pointer is updatred. When the vocabulary is referenced when the user variable CONTEXT is set to contain the address of the pointer field within the vocabulary word with the result that all dictionary searches for word names begin with the latest entry in that vocabulary. The word DEFINITIONS copies the pointer field address in CONTEXT to the user variable CURRENT with the result that newly defined words will be added to that vocabulary.

The core language consists of a single vocabulary called FORTH. Until other vocabularies are added the latest entry in this vocabulary is also the last word in the dictionary. A new vocabulary is added using the vocabulary defining word in a sequence such as VOCABULARY EDITOR which creates a new vocabulary called EDITOR. The sequence EDITOR DEFINITIONS allows words to be added to the new vocabulary. When the editor is loaded the dictionary contains two words with the name I. The sequence EDITOR I gives I the meaning insert text from PAD while FORTH I gives I the meaning of copy the index inside a DO LOOP construct to the stack.

By convention vocabulary words are made immediate so the complete defining sequence for the vocabulary word EDITOR is VOCABULARY EDITOR IMMEDIATE. This means that both the meanings of I can be used in the definition of another word. If the vocabulary word itself is to be part of a definition then the sequence [COMPILE] EDITOR must be used to force compilation.

Whenever the word FORGET is used to discard words from the dictionary the CONTEXT and CURRENT vocabularies must be the same. DEFINITIONS may be used to set the CURRENT vocabulary to the CONTEXT vocabulary.

Formatted number output

The number output operators are used to convert a binary number to a string of ASCII characters and format the result with spaces and other non numeric characters as required. The string is built up at PAD and is usually output by TYPE when it is complete.

A number is converted to digits by repeatedly dividing by the current value of BASE. The remainder of each division forms one digit of the number starting with the least significant digit. Consequently the character string is built up from right to left. Other characters are inserted as required. The number actually converted is always an unsigned double precision value. Single

precision numbers must always be converted to double precision first. A sign is added in a separate operation.

The conversion process is started by <# whcih initialises the user variable HLD. HLD contains a pointer to the left most character in the string which grows downwards in memory from PAD. The word # converts one digit from a double precision number on the stack by dividing it by the value of BASE. The double precision quotient is left on the stack and the remainder is converted to an ASCII character and inserted in the character string. The word #s performs the action of # repeatedly until no more significant digits can be generated and a double precision zero is left on the stack. Both # and #S always produce at least one digit even if it is zero.

The word SIGN is used to test the sign of a signed number and add a minus sign to the string if the number is negative. SIGN uses the third stack entry, immediately under the double precision number being converted, which must be a copy of the high order part of the double precision number made before DABS is used to make the number unsigned for conversion into digits.

Other characters can be inserted into the string when required by the sequence c HOLD which inserts the character c. Finally #> is used to complete the process by dropping the double precision number on the stack and leaving the address and character count of the character string for use by TYPE.

```
: UD.           <# #S #> TYPE SPACE ;
: D.            SWAP OVER DABS
                <# #S SIGN #> TYPE SPACE;
```

In the first example UD. outputs an unsigned double precision number from the stack using #S to generate all the significant digits with no other characters. The second shows a possible definition of D. (which is in the core language). The sequence SWAP OVER DABS makes a copy of the high order part of a double precision number on the stack in the third stack entry position before DABS forms the absolute value of the double precision number. The third stack entry is used by SIGN, which adds a minus sign to the character string if it is negative.

Single precision numbers must be converted to double precision before they can be operated on by the number formatting words.

```
: U.            0
                <# #S #> TYPE SPACE ;
: .             DUP ABS 0
                <# #S SIGN #> TYPE SPACE ;
```

U. outputs an unsigned single precision number which is converted to double precision by pushing a zero in front of it on the stack. In the signed word . a copy of the number is made as the third stack entry for use by SIGN. The sequence DUP ABS 0 is

more economical than S->D SWAP OVER DABS.

These examples show the basic sequences needed to output single and double precision signed and unsigned numbers. Below are two more sophisticated examples of formatting.

```
HEX

: ZEROS         0 MAX -DUP
                IF 0 DO 30 EMIT LOOP ENDIF ;
: UD.ZR         >R
                <# #S #>
                R> OVER - ZEROS TYPE ;
: SEX           # 6 BASE ! # DECIMAL ;
: .TIME         <# SEX 3A HOLD SEX 3A HOLD # # #>
                TYPE SPACE ;

DECIMAL
```

UD.ZR outputs a double precision number occupying the second and third stack entries right adjusted in a field of zeros whose width is the first stack entry. Thus it behaves similarly to D.R. In UD.ZR the field width is kept on the return stack while number conversion takes place. After number conversion ends with #> the count and address of the character string representing the number is left on the stack. The field width is restored to the parameter stack by R> and OVER - copies the count over the field width and subtracts it from the field width. The difference between the two is used by ZEROS to fill the unused part of the field with zeros from the left and then the character string is added by TYPE.

ZEROS takes one value from the stack and outputs that number of zeros. Nothing is output if the number is less than one. Note that -DUP duplicates the first stack entry only if it is non zero and that IF is used to make an implicit test for a non zero value. Because numbers output by . are always followed by a space the ASCII sequence 30 EMIT rather than 0 . has to be used inside the loop to output each zero.

The word .TIME takes a double precision number from the stack representing seconds and converts it to an output of hours minutes and seconds in the format 00:00:00. For the seconds and minutes (remembering that the character string is built up from right to left) .TIME uses SEX to convert one digit in the decimal base followed by one in base 6. Between the pairs of digits a colon is inserted by 3A HOLD where 3A is the ASCII code for : . Two hours digits are converted in the decimal base.

Characters and Strings

Single characters are usually placed in stack entries where the ASCII value of the character occupies the lower seven or lower eight bits of a sixteen bit stack entry. A character can be output from the stack using EMIT. The sequence c EMIT outputs

the character whose ASCII value is c to the screen. EMIT, which is used for all output of characters, increments a user variable named OUT each time it is used. The value in OUT can be used to control output formatting.

There are three predefined words for outputting particular characters. SPACE and CR output a space and carriage return respectively and require no stack entries. SPACES takes one stack entry and outputs that number of spaces.

A character can be placed on the stack by specifying its ASCII value as a number. A character may also be obtained from the keyboard by the word KEY which waits for a key to be pressed and places the ASCII value on the stack.

Strings are not usually themselves placed on the stack. Instead a string is stored in memory and its address and sometimes a count of its characters manipulated in the stack. Strings are usually read from the normal input text using WORD. The sequence c WORD scans the input text until the delimiter character c or the end of text marker ASCII zero is found. Leading examples of the delimiter character are ignored. The text enclosed in this way is moved to the dictionary pointer preceded by a byte containing a count of the number of characters. The dictionary pointer and the free area of memory immediately above it is a convenient temporary location and its address is placed on the stack by HERE.

The text interpreter of the language itself uses WORD to intrerpret text typed at the keyboard or stored on screens. The delimiter is usually a space. Thus for general purposes it is often desirable to move the string away from the dictionary pointer to anotyher temporary location for further processing. This is provided by PAD which returns an address 64 bytes after the current value of the dictionary pointer.

There are a number of words for manipulating memory regions that are specially useful for text processing. The sequence address count byte FILL fills an area of memory with the specified byte starting at the address and for the number of locations given by count. Two special cases of FILL are address count ERASE and address count BLANKS which fill a memory region with zeros and spaces (ASCII 20 hex) respectively. A fourth important word is CMOVE. The sequence source destination count CMOVE moves the number of bytes given by count from the source address to the destination address. The bytes with the lowest memory address are mopved first and care must be taken if the source and destination areas overlap.

An example of the use of these words is shown below in the word TEXT which is on the first editor screen.

```
: TEXT          HERE C/L 1 + BLANKS
                WORD
                HERE PAD C/L 1+ CMOVE ;
```

The sequence c TEXT scans the input text stream and moves text enclosed by the delimiter c to PAD. The constant C/L returns the number of characters per line. The sequence HERE C/L 1+ BLANKS fills an area of memory starting at the dictionary pointer and one more than C/L bytes long with spaces. Then WORD takes the ASCII character from the stack and encloses text placing at the address of the dictionary pointer. The sequence HERE PAD C/L 1+ CMOVE moves the string to PAD.

The usual space must be left between TEXT (or a word which uses TEXT) and the string being read. At the end of the string must be the delimiter character. For example BL TEXT where BL is a constant returning an ASCII space will enclose text up to the next space while HEX 22 TEXT will enclose text up to the next quotation marks " whose ASCII value is 22 hex. To enclose all the text on a line the convention is to use 1 TEXT since an ASCII 1 is unlikely to be in the text being scanned.

Once a string is at PAD it can be processed further or stored in a string variable or in the blocks of mass storage or wherever it is required. A simple defining word for a string variable is below.

```
:  $VARIABLE     <BUILDS HERE C/L 1+ BLANKS C/L 1+ ALLOT
                 DOES> ;
$VARIABLE TEST
```

The action of <BUILDS and DOES> is explained in detail in the subsection on high level defining words. TEST behaves like an ordinary variable in that it returns an address. Instead of having two bytes to store a number it has C/L 1+ bytes to store a string because the defining word $VARIABLE contains the sequence C/L 1+ ALLOT. These bytes are initialised to spaces by the sequence HERE C/L 1+ BLANKS. A string can be moved from PAD to TEST by PAD TEST C/L 1+ CMOVE.

On a disc system the blocks of mass storage are usually the best place to store strings. Otherwise it may be worth developing more sophisticated versions of $VARIABLE.

Strings and text in general can be output using TYPE. The sequence address count TYPE outputs a string of n characters where n equals the count stored at the address. Note that only the characters of the string themselves are in memory this time.

If the string is in the frequently found and very useful format of a byte containing a count of the characters in the string followed by the characters then the sequence addr COUNT TYPE will output the string. This time the address is that of the byte containing the count and COUNT will form the correct arguments on the stack for TYPE.

Frequently there are several spaces after the last significant character in a string and these can be eliminated by -TRAILING. This word is used immediately before TYPE in a sequence such as

address count -TRAILING TYPE and adjusts the count to avoid outputting spaces unnecessarily. The string itself is not altered.

The protocol used by FORTH is sometimes described as user driven. There is no prompting as such and the system waits for input instructions which are carried out and then simply waits for further instructions. Numbers required as parameters precede and text follows the words that use them. This arrangement is very general and a highly desirable attribute of a self extending language. An individual program may require a prompting system and close control of input. Like most things in FORTH this can be provided.

The word QUERY accepts a line of up to eighty characters to the terminal input buffer whose address is available in the user variable TIB. A line in this buffer can be processed easily with WORD. The sequence address count EXPECT accepts a line of characters from the keyboard with a maximum number of characters equal to count and terminated in any case by RETURN. The characters are stored at the address and the end of the line is marked by three ASCII zeros. EXPECT, which is used by QUERY and the language itself for accepting input, is somewhat primitive recognising only backspace as a control key and treating all other keys as data. A more sophisticated version of EXPECT could be written to user requirements even for general purpose use.

With these words it is possible to define input words similar to those found in other languages.

```
: GET$          QUERY 1 TEXT ;
: GET#          QUERY 1 TEXT PAD NUMBER ;
```

The word GET$ awaits the input of a string and moves it to PAD. It could be included in a sequence with a prompt such as CR ." Enter answer " GET$.

It is often desirable to be able to convert strings of characters to numbers. This can be achieved by NUMBER which is used in a sequence such as address NUMBER and converts a string stored at the address (including a byte with a character count) to a double precision number with reference to the current base. An invalid character will result in error MSG 0. An example of the use of this word is shown in GET# which returns a double precision number.

Colon Compiler

The text interpreter, which interprets the text typed at the keyboard and stored on screens, has two modes. In the execute mode words are searched for by name in the dictionary and then executed and numbers are pushed to the stack. When : (colon) is used to start the definition of a new word the text interpreter enters its compile mode. In this mode it generates the threaded code which represents compiled Forth. Words are still searched

for by name in the dictionary but are not executed; instead their addreses are added to the definition of the new word being added to the dictionary. Numbers are also written into the threaded code preceded by a literal handler which will push the number to the stack when the code is executed.

The word : forms a dictionary entry for the name of the new word it defines. It also places the text interpreter in the compile mode by setting the value of the user variable STATE to CO hex. The word ; completes a definition by, among other actions, setting the value of STATE to zero which restores the execute mode. Inbetween the two words and numbers are compiled and not executed unless instructions are given otherwise. This subsection is concerned with those instructions and controlling the compilation process.

There are some words whose function is to control compilation itself. These words are called immediate words and they are executed and not compiled in the compile mode. The word ; (semicolon) is an immediate word because it executes to terminate the definition process rather than itself being compiled. A word is made an immediate word by writing the word IMMEDIATE after its definition.

By convention vocabulary words are made immediate so that different vocabularies can be selected during a definition. To include a vocabulary word in a definition it must be compiled and not executed even though it is an immediate word. This is achived by [COMPILE] which forces compilation of the word which follows it even if it is immediate. Thus including [COMPILE] EDITOR as part of a definition would make the vocabulary word EDITOR part of the new word being defined rather than select the editor vocabulary during the definition.

A second variation on this theme is to cause words which are not immediate words to be executed and not compiled inside a colon definition. This is done by enclosing them in the square brackets [and]. [is an immediate word which sets the value of STATE to zero thus placing the text interpreter in the execute mode;] places the text interpreter in the compile mode. For example the sequence [HEX] will change the number base to hex during the definition. HEX does not become part of the definition.

An important use of [and] is the evaluation of expressions at compile time. If the sequence 27 3/ was part of the definition of a word then the division would be carried out when the word was executed. If the sequence is changed to [27 3/] LITERAL then the division will be carried out when the word is compiled and the result compiled as a number literal. The sequence [27 3/] enters the execute mode, forms the result of the expression on the stack and returns to the compile mode. The immediate word LITERAL compiles the first stack entry as a number literal. This process can be extended to less trivial expressions. Each component of the expression must, of course, be a constant.

The definition of LITERAL itself provides the final example, in which one word, usually an immediate word, compiles another when it is executed.

```
: LITERAL       STATE @ IF COMPILE LIT , ENDIF ;   IMMEDIATE
```

LITERAL first tests the value of STATE and does nothing if the text interpreter is in the execute mode. The word COMPILE will cause the word that follows it, in this case the number literal handler LIT, to be compiled when LITERAL is executed. The word , (comma) will put the number on the stack when LITERAL is executed into the compiled code after the code for LIT. Thus in the compile mode LITERAL will compile the first stack entry as a number literal.

A word in Forth can include itself in its definition. This process is called recursion and allows the computation of functions which can be defined in terms of themselves. For example the factorial function can be defined as:

```
factorial (1) = 1
factorial (n) = n * factorial (n-1)
```

This can be written in Forth as:

```
:  CALL         LATEST PFA CFA , ;          IMMEDIATE
:  FACTORIAL    DUP 2 <
                IF DROP 1
                ELSE DUP 1 - CALL *
                ENDIF ;
```

FACTORIAL is used in a sequence such as n FACTORIAL and returns n factorial on the stack. The immediate word CALL obtains the code field address of FACTORIAL and uses comma to add it to the compiled code. FACTORIAL cannot be referred to by name while it is being defined because the definition is not complete. Although the factorial of a number less than one is undefined, the definition shown will return to one, providing a safe exit if a number less than one is given as a parameter.

Floating Point Extension

The floating point extension is designed to be as closely integrated as possible with the rest of the language and follows the same conventions and procedures as single and double precision numbers. The floating point operators have their own vocabulary and are made available be the vocabulary name FLOATING.

It will be recalled that a single precision integer is written as a sequence of digits with no other characters and that a double precision integer is written as a series of digits including a point. Floating point numbers are written as a series of digits including a comma and may include an exponent preceded by E.

Only the decimal base may be used for the input (and output) of floating point numbers. Below are some examples of the different types of number.

single precision integer	0	1	-34	31345	
double precision integer	0.	1.	-34.	200000.	
floating point	0,0	1,0	-34,26	2,0E5	2,0E-5

Up to nine significant decimal figures may be used. Numbers with a magnitude between 1,4693680E-39 and 1,7014118E38 can be represented together with zero. A floating point number can be printed by F. which generates eight significant figures after rounding the ninth figure plus an exponent, in standard form.

Each floating point number occupies three stack entries of six bytes. The words FDUP FSWAP FDROP FOVER and F2DUP allow the numbers to be manipulated on the stack; they are equivalents of the single precision stack operators. The basic arithmetic operations are performed by F+ F- F* and F/ while FABS returns the absolute value of a floating point number. The defining words FCONSTANT and FVARIABLE allow floating point constants and variables. The value of a variable is obtained from the address its name leaves on the stack by F@ and a new value stored in it by F!.

The word FLOAT allows a double precision integer and a single precision integer decimal exponent to be converted to a floating point number. For example 123. -2 FLOAT will form the floating point number 1,230E0. Inside a DO LOOP construct the index, placed on the stack as a single precision integer by I, can be converted to a floating point number using S->D to convert it to double precision and FLOAT.

```
: FCOUNT        FLOATING 10 0 DO I S->D  0 FLOAT  CR F.  LOOP ;
```

The vocabulary word FLOATING must be used inside the definition to select the FLOATING vocabulary because colon automatically sets the context vocabulary equal to the current vocabulary.

Floating point numbers can be converted to double precision integers using INTEGER which truncates any fractional part.

A full set of relational operators allow tests and comparisons involving floating point numbers. The flags generated can be used by the usual BEGIN UNTIL and BEGIN WHILE REPEAT constructs and the IF ELSE ENDIF construct.

Each floating point number is made up of a single precision exponent representing a power of two and a double precision fractional part. The latter is a signed twos complement number with an imaginary binary boint after the sign bit which is the most significant bit. It is also shifted as far to the left as possible and the exponent adjusted appropriately. The whole number is the product of the fractional part and two raised to a power equal to the exponent. The internal accuracy is governed by

the fractional part and is the same as that for double precision integers of slighly better than nine decimal figures. It should be remembered though that certain simple numbers (0,1 for example) become irrational when expressed in binary. The range is governed by the exponent which must lie between -128 and +127 and internally is a byte length signed twos complement number.

Overflow is not reported mainly because of the nature of the twos complement binary exponent. A calculation may overflow internally to a considerable extent but give a correct result on output provided the final answer is within range. As with integer arithmetic division by zero returns the maximum possible value with the appropriate sign.

```
FLOATING DEFINITIONS
: F+!           DUP >R F@ F+  R> F! ;
: FMINUS        >R  DMINUS  R> ;
```

In the two examples above FLOATING DEFINITIONS has made FLOATING the current vocabulary so FLOATING is not needed inside the definitions. The word F+! behaves similarly to +! and can be used to increment the value of a floating point variable. When F+! is used in a sequence such as 1,35 TOTAL F+! where TOTAL is a floating point variable one copy of the address of the variable is kept on the return stack and the addition performed on the parameter stack. In the second example the binary exponent is stored on the return stack while the sign of the fractional part is reversed using the double precision operator DMINUS.

Floating point numbers are a comparatively little explored area of Forth. Integers should not be neglected though and should continue to be used for counting, loop control and those applications where the speed and accuracy of integer operations are required.

The output of floating point numbers may be formatted. The paragraphs below should be read in conjunction with the subsection entitled Formatted Number Output which describes the process for integer numbers.

```
: OUTPUT        SWAP OVER DABS
                <# BASE @ M/MOD ROT DROP
                7 0 DO # LOOP 2C HOLD
                #S SIGN #> TYPE ;

: F             (F.)
                >R OUTPUT R>
                ." E" . ;
```

The example above shows how the number formatting operators may be used to output floating point numbers in the format 0.0000000E0. The principles employed may be used to produce any desired format.

In the word F. the operator (F.), which is a primitive in the FLOATING vocabulary, converts a floating point number on the stack to a double precision integer greater than or equal to 1E8 but less than 1E9 and a single precision integer decimal exponent.

The double precision integer has two special features. It has been rounded in the ninth significant figure and there is an implied decimal point between the first and second significant figures. The word OUTPUT shows how this number may be formatted. Its range indicates that it contains nine decimal digits and each of these must extracted although some may then be discarded. Thus in the example the ninth least significant figure is discarded by the sequence BASE @ M/MOD ROT DROP. More figures could be discarded by placing this sequence in a loop. Seven figures are added to the ASCII string by the sequence 7 0 DO # LOOP. Finally, after inserting the decimal comma, the most significant figure is added to the string by S#. The total number of figures dealt with must always add up to nine.

Returning to F. the decimal exponent is stored on the return stack while the rest of the number is processed by OUTPUT. The decimal exponent is a single precision integer whose value has been adjusted to reflect the implied positon of the decimal point. In this example it is output simply by DOT but it could be formatted differently if desired.

Floating Point Arithmetic Operators

Word	Stack		Action
F+	fn1 fn2	fn1+fn2	Adds two floating point numbers.
F-	fn1 fn2	fn2-fn1	Subtracts the first floating point number from the second.
F*	fn1 fn2	fn1*fn2	Multiplies two floating point numbers.
F/	fn1 fn2	fn2/fn1	Divides the second floating point number by the first.
FABS	fn	abs(fn)	Returns the absolute value of a floating point number.
F.	fn		Prints the first floating point number on the stack in standard form. The result is valid only in the decimal base.
FDUP	fn1	fn1 fn1	Duplicates a floating point number on the stack.

FSWAP	fn1 fn2	fn2 fn1	Exchanges two floating point numbers.
FDROP	fn		Drops a floating point number.
FOVER	fn1 fn2	fn2 fn1 fn2	Copies the second floating point number over the first.
F2DUP	fn1 fn2	fn1 fn2 fn1 fn2	Duplicates a pair of floating point numbers.
F0=	fn	f	Returns a flag which is true if the number equals zero.
F0<	fn	f	Returns a flag which is true if the number is less than zero.
F>	fn1 fn2	f	Returns a flag which is true if fn2 is greater than fn1.
F>=	fn1 fn2	f	Returns a flag which is true if fn2 is greater than or equal to fn1.
F<	fn1 fn2	f	Returns a flag which is true if fn2 is less than fn1.
F<=	fn1 fn2	f	Returns a flag which is true if fn2 is less than or equal to fn1.
FCONSTANT	fn		fn FCONSTANT cccc creates a floating point constant named cccc value fn. When the constant is used it leaves its value as a floating point number on the stack.
FVARIABLE	fn		fn FVARIABLE cccc creates a floating point variable named cccc initial value fn. When the variable is used it leaves its address on the stack.
F@	addr	fn	Replaces an address with the floating point number stored at the address.
F!	addr fn		Stores the floating point number at the address.
FLOAT	n d	fn	Converts a double precision number and a single precision decimal exponent to a floating point number.

Word	Stack		Action
INTEGER	fn	d	Converts a floating point number to a double precision integer truncating any fractional part.
FLITERAL	fn		In the compile mode compiles a floating point number as a literal. An immediate word usually used inside a colon definition. In the execute mode takes no action.
NUMBER	addr	n d	Converts a string consisting of a count and that number of bytes stored at the address to a double precision integer and single precision decimal exponent. If the value of the variable FPL is -1 the decimal exponent has no significance; otherwise use FLOAT to form a floating point number.
FPL	addr		A variable whose value is -1 if the last number input was not a floating point number.
INTERPRET			Text interpreter which performs the same function at INTERPRET in the FORTH vocabulary except that it also processes floating point numbers.
(F.)	fn	n d	Converts a floating point number to a double precision integer between 1E8 and 1E9 and a decimal exponent. The integer is rounded in the ninth significant decimal figure and the exponent corrected for output in standard form.
FLOATING			Selects floating point vocabulary. Like all vocabulary words FLOATING is immediate.
OUTPUT	d		Special purpose number formatting word which formats the double precision integer formed by (F.). Used by F..
-ZEROS	n addr	n addr	Adjusts parameters used by TYPE to eliminate trailing zeros from a decimal number in standard

Word	Stack		Action
			form.
(NUMBER)	addr d	addr d	Converts ASCII characters stored at addr +1 to digits and accumulates them into the double precison number d. Counts the number of digits processed into variables DPL and PPL unless their value is -1. Returns the address of the first unconvertable character.
PUNCT EXPON			Special purpose words used by NUMBER.
PICK	n	nth stack entry	n PICK copies the nth single precision stack entry. Thus 1 PICK is the same as DUP and 2 PICK as OVER. Used for manipulating mixed floating point and other numbers on the stack.

String Handling Words

This description of the three screens of string handling words which are on the disc or tape in souce form should be read in conjunction with the subsection Characters and Strings in Forth Techniques. The object of these words is to show one way in which the string handling techniques found in some other languages can be implemented in Forth.

The format chosen for strings is that of a byte containing the length of the string followed by the characters in the string. The whole is stored in memory and the addresses of strings, which are the addresses of their length bytes, are manipulated on the stack. Strings are never themselves placed on the stack although single characters may be.

There are two temporary locations for storing and procesing strings. These are PAD and 2PAD defined as 68 and 136 bytes after the dictionary pointer respectively. It is important to note that these locations move when the dictionary pointer moves and that use should be made of string variables or similar arrangements for the permanent storage of strings. PAD and 2PAD are, none the less, adequate for most applications as temporary stores.

Strings should be placed between " and " with one space at least between the first " and the start of the string. When " is used inside a word definition the string is compiled and placed at PAD when the word is executed. Outside a word definition, in the

execute mode, " simply places the string at PAD.

A string is printed by $. which requires an address, that of the length byte of the string, on the stack. Thus a string at PAD is printed by PAD $. .

The word >2PAD copies the string at PAD to 2PAD. In general strings are copied using CMOVE in a sequence such as source address destination address number of bytes to be moved CMOVE. Avoid overlapping moves; move a string to a third location first. There are no run time checks on inappropriate uses of CMOVE.

LEFT$ is used in a sequence such as n LEFT$. It operates on the string at PAD and replaces it with its left substring of n characters and uses 2PAD. n RIGHT$ performs the analogous operation forming the right substring. n1 n2 MID$ forms a substring n1 characters from the left of the string at PAD n2 characters long provided sufficient characters are available.

Strings can be compared using $= which requires two addresses, one for each string, on the stack. addr1 addr2 $= returns a flag which is true only if the two strings are identical.

String Operators

Word	Stack	Action
(")		The run time procedure which transfers a string embedded in compiled code to PAD.
"		Encloses a string up to the delimiter ". In the execute mode moves the string to PAD; in the compile mode compiles the string.
$VARIABLE		Defining word used in a sequence such as $VARIABLE cccc which creates a string variable named cccc initialised to blanks. When the variable is used leaves the address of the string space on the stack.
2PAD	addr	Temporary location 136 bytes after current value of dictionary pointer.
BLANKPAD		Fills PAD with blanks.

>2PAD			Copies the string at PAD to 2PAD.
LEFT$	n		Replaces the string at PAD with its left substring of n characters.
RIGHT$	n		Replaces the string at PAD with its right substring of n characters.
MID$	n1 n2		Replaces the string at PAD with a substring starting n1 characters from the left, n2 characters long.
$=	addr1 addr2	f	Returns a true flag in the strings whose addresses are on the stack are identical. Otherwise returns a false flag.
$.	addr		Outputs a string stored at the address.
MATCH	n1 addr 1 n2 addr2	n f	Searches forward from addr2 for n2 bytes seeking a match for a string of n1 characters whose first character (not its length byte) is at addr1. Returns pointer to immediately after the matched string as an offset from addr2 and a flag which is true only if a successful match is made.

GLOSSARY

STACK OPERATORS

SINGLE PRECISION ARITHMETIC OPERATORS

DOUBLE PRECISION ARITHMETIC OPERATORS

MIXED PRECISION ARITHMETIC OPERATORS

LOGICAL OPERATORS

OUTPUT OPERATORS

ADDRESS OPERATORS

RELATIONAL OPERATORS

CONDITIONAL AND LOOPING OPERATORS

RETURN STACK OPERATORS

DICTIONARY AND MEMORY MANAGEMENT WORDS

DEFINING WORDS

CHARACTER AND STRING OPERATORS

NUMBER FORMATTING OPERATORS

VOCABULARY CONTROL WORDS

USER VARIABLES

SYSTEM CONSTANTS

SYSTEM OPERATORS

VIRTUAL MEMORY OPERATORS

ERROR CONTROL WORDS

EDITOR WORDS

Stack Operators

NOTE the top stack entry is on the LEFT

Word	Stack		Action
DROP	n		Discards the first stack entry.
DUP	n	n n	Duplicates the first stack entry.
-DUP	n	n n	Duplicates the first stack entry only if it is not equal to zero.
SWAP	n1 n2	n2 n1	Reverses the order of the first two stack entries.
OVER	n1 n2	n2 n1 n2	Duplicates the second stack entry over the first.
ROT	n1 n2 n3	n3 n1 n2	Rotates the third stack entry to the top of the stack.
2DUP	d	d d	Duplicates a double precision number or a pair of 16 bit numbers.
SP@		addr	Returns the value of the stack pointer which points to the first stack entry as it was before SP@ was used.
SP!		empty	Clears the stack by initialising the stack pointer

Single precision arithmetic operators

Word	Stack		Action
+	n1 n2	n1+n2	Adds first two stack entries
-	n1 n2	n2-n1	Subtracts first stack entry from second
*	n1 n2	n1*n2	Multiplies first two stack entries

/	n1 n2	n2/n1	Divides second stack entry by first
MOD	n1 n2	mod n2/n1	Forms second stack entry modulo the first
/MOD	n1 n2	n1=quot n2=rem	Divides second stack entry by first. The quotient is the first stack entry and the remainder is returned as the second.
*/	n1 n2 n3	n1 = result	Multiplies the second stack entry by the third forming a 32 bit product which is divided by the first stack entry. result = n2*n3 /n1
*/MOD	n1 n2 n3	n1 = result n2 = rem	As */ except that the remainder from the division is returned in addition to the quotient.
ABS	n1	abs (n1)	Returns the absolute value of the first stack entry.
MINUS	n1	-n1	Reverses the sign of the first stack entry.
MAX	n1 n2	n1 = larger	Returns larger of first two stack entries.
MIN	n1 n2	n1 = smaller	Returns smaller of first two stack entries.
1+	n1	n1 + 1	Increments the first stack entry.
2+	n1	n1 + 2	Increments the first stack entry twice.
S->D	n1	d1	Converts the signed single precision entry to signed double precision.
+-	n1 n2	n1 = result	If n1 is negative reverses the sign of n2. n1 is discarded and n2 is the result.

Double Precision Arithmetic Operators

Word	Stack		Action
D+	d1 d2	d1+d2	Adds two double precision numbers.
DABS	d1	abs(d1)	Returns the absolute value of a double precision number.
DMINUS	d1	-d1	Reverses the sign of a double precision number.
D+-	n d	d = result	If n is negative reverses the sign of d. n is discarded.

Mixed Precision Arithmetic Operators

Word	Stack		Action
M*	n1 n2	d = n1*n2	Multiplies two sixteen bit numbers to give a 32 bit result.
M/	n d	n1 = d/n n2 = rem	Divides a 32 bit number by a 16 bit number to give a signed 16 bit quotient and a signed 16 bit remainder.
M/MOD	n d	d = d/n n = rem	Divided an unsigned 32 bit number by an unsigned 16 bit number. Returns a 32 bit quotient and a 16 bit remainder.
U*	u1 u2	ud = u1*u2	Multiplies two unsigned 16 bit numbers to give an unsigned 32 bit result.
U/	u ud	u1 = ud/u u2 = rem	Divides an unsigned 32 bit number by an unsigned 16 bit number to give a 16 bit quotient and a 16 bit remainder.

Logical Operators

Word	Stack		Action
AND	n1 n2	n1 = n1 AND n2	Logically ANDs the first and second stack entries bit by bit.
OR	n1 n2	n1 = n1 OR n2	Logically ORs the first and second stack entries bit by bit.
XOR	n1 n2	n1 = n1 XOR n2	Logically ORs the first and second stack entries bit by bit.
0=	n1	NOT n1	Performs the logical NOT function reversing the truth state of the first stack entry

Output Operators

Word	Stack	Action
		a signed 16 bit twos complement number in the current number base followed by one space.
U.	u1	Print the first stack entry as an unsigned 16 bit number in the current number base followed by one space.
D.	d1	Print a signed 32 bit twos complement number in the current number base followed by one space.
?	addr	Print the contents of an address as a signed 16 bit number followed by one space.
.R	n1 n2	Displays the second stack entry right adjusted in a field whose width is the first stack entry. The second stack entry is printed as a signed number.
D.R	n1 d1	Displays the double precision signed number in a field whose

			width is the first stack entry.

Address Operators

Word	Stack		Action
@	addr	n1	Replaces an address with the 16 bit contents of the address. Pronounced fetch.
!	addr n1		Stores a 16 bit number at the address. Pronounced store.
+!	addr n1		Adds a 16 bit number to the 16 bit contents of the address and leaves the result at the address. Pronounced plus store
C@	addr	b1	Replaces an address with its 8 bit contents setting them in the low order 8 bits of a 16 bit stack entry. Pronounced c fetch.
C!	addr b1		Stores the low order 8 bits of a 16 bit stack entry at the address. Pronounced c store.
2@	addr	d1	Replaces an address with its 32 bit contents. Pronounced two fetch.
2!	addr d1		Stores a 32 bit number at the address. Pronounced two store

Relational Operators

Word	Stack		Action
=	n1 n2	f	Returns a true flag if the first two stack entries are unequal; otherwise returns a a false flag.
-	n1 n2	f	Returns a true flag if the first two stack entries are unequal; otherwise returns a

			false flag. The true flag is equal to the difference between n1 and n2.
>	n1 n2	f	Returns a true flag if n2 is greater than n1. Otherwise returns a false flag.
<	n1 n2	f	Returns a true flag if n2 is less than n1. Otherwise returns a false flag.
U<	u1 u2	f	Returns a true flag if the unsigned number u2 is less than the unsigned number u1. Otherwise returns a false flag.
0=	n1	f	Returns a true flag if n1 is equal to zero. Otherwise returns a false flag.
0<	n1	f	Returns a true flag if n1 is less than zero and thus a negative number. Otherwise returns a false flag.

Conditional and Looping Operators

Word	Stack	Action
IF ...	f	IF takes a flag from the stack. The words after IF are executed only if the flag is true. The words in the optional ELSE clause are executed only if the flag is false. The words after ENDIF are executed unconditionally.
ELSE ...		
ENDIF		
THEN		An alias for ENDIF.
DO ...	index limit	Defines a finite loop. The index is incremented by one on each passage through the loop. The loop ends when the limit is reached or passed.
LOOP		
DO ...	index limit	Defines a finite loop. The index is incremented by the value n taken from the stack by +LOOP on each passage
+LOOP	n	

			through the loop. If the increment is positive the loop ends when the index is equal to or greater than the limit. If the increment is negative the loop ends when the index is equal to or less than the limit.
LEAVE			Terminates the execution of a finite loop at the next LOOP or +LOOP.
BEGIN ...			Defines an indefinite loop which is repeated until the flag taken from the stack by UNTIL is true.
UNTIL	f		
END	f		An alias for UNTIL.
BEGIN ...			Defines an indefinite loop which is repeated until the flag taken from the stack by WHILE is false.
WHILE ...	f		
REPEAT			
BEGIN ...			Defines an unconditional loop.
AGAIN			

Return Stack Operators

Word	Stack		Action
R>		n	Removes one value from the return stack and places it on the parameter stack.
>R	n		Removes one value from the parameter stack and places it on the return stack.
I		n	Inside a DO LOOP construct copies the loop index (which is the top return stack value) to the parameter stack. If there is more than one nested loop then it copies the index

		of the innermost loop.
R	n	Copies the first return stack value to the parameter stack.
RP@	n	Places the value of the return stack pointer on the parameter stack.
RP!		Clears the return stack by reinitialising the return stack pointer.

Dictionary and Memory Management Words

Word	Stack	Action
ALLOT	n1	Reserves n bytes in the dictionary by advancing the dictionary pointer.
C,	b	Stores the lower 8 bits of the first stack entry in the next free dictionary byte and advances the dictionary pointer one byte.
,	n1	Stores the first stack entry at the next free dictionary location and advances the dictionary pointer.
DP	addr	User variable containing the address of the next free dictionary location referred to as the dictionary pointer.
HERE	addr	Leaves the value of the dictionary pointer which is the address of the next free dictionary location on the stack.
BLANKS	n addr	Fills an area of memory starting at addr for n bytes with spaces (ASCII 20 hex).
ERASE	n addr	Fills an area of memory starting at addr for n bytes with zeros.
FILL	b n addr	Fills an area of memory starting at addr for n bytes

		with the byte b.
CMOVE	n dest source	Moves n bytes from source to dest where source and dest are addresses. The bytes are moved in order of increasing memory address.

Defining Words

Word	Stack	Action
CREATE		CREATE cccc creates a dictionary header named cccc with a code address pointing to the parameter field. Used by other defining words to form all dictionary headers.
CONSTANT	nl	n CONSTANT cccc creates a constant named cccc value n . When the constant is used it leaves its value on the stack.
VARIABLE	nl	n VARIABLE cccc creates a variable named cccc initial value n . When the variable is used it leaves its address on the stack.
USER	nl	n USER cccc creates a user variable named cccc whose address is an offset n from the user variable base register value. When the user variable is used it leaves its address in the user variable area.
<BUILDS		Used in the colon definition of defining words. Directs defining word to create dictionary entry. Always used with DOES>. See High Level Defining Words.
DOES>		Used in colon definition of defining words. Controls run time behaviour of the class of words defined by the defining word. Always used with <BUILDS. See High Level Defining Words.

:	: cccc creates a new word named cccc . All words and numbers entered until ; or ;CODE are compiled into the definition unless they are immediate words which are executed. : sets the context vocabulary to current.
;	Terminates a definition started by : .
;CODE	Ends compilation of the high level part of a machine code defining word by compiling (;CODE).
(;CODE)	Runtime procedure compiled by ;CODE which stores a pointer to the machine code part of the defining word in the code field of the latest entry in the dictionary.
;S	Runtime procedure compiled by ; at the end of a colon definition which returns control to the calling procedure when the word is executed.
COMPILE	Compiles the execution address of the word following COMPILE into the dictionary.
[COMPILE]	Forces compilation of an immediate word
[	Suspends compilation within a colon definition and allows a sequence of words to be executed to make a calculation for example.
]	Resumes compilation in a colon definition.
LITERAL n1	When compiling compiles the first stack entry as a 16 bit literal. LITERAL is immediate and is usually used in colon definitions. In the execution mode it takes no action.

DLITERAL	d1		Performs the action of LITERAL for a double precision number.

Character and String Operators

Word	Stack		Action
			literal cccc into the definition of a word. In the execution mode the string is printed immediately.
(.")			The runtime procedure compiled by ." which outputs an embedded string.
COUNT	addr	n addr	Converts an address which points to a string stored in memory consisting of a byte containing the length of the string followed by bytes containing the characters to n which is the number of bytes in the string and addr which points to the first character. These stack entries are usually used by TYPE.
CR			Outputs a carriage return and line feed.
EMIT	c		Outputs the character whose ASCII code is c . The user variable OUT is incremented to allow control of formatted output.
EXPECT	n addr		Accepts characters from the keyboard and stores them beginning at the address until return or until n characters have been received. Two nulls or ASCII zeros mark the end of the text.
MEXPECT	n addr		Similar to EXPECT. However does not return until RETURN or CNTRL-STOP pressed. Updates MEXIT. Responds to cursor movements.
MEXIT		a	Variable updated by MEXPECT. Contains non-zero if CONTROL-

			STOP pressed.
KEY		c	Waits for a key input and returns the ASCII value.
NUMBER	addr	d	Converts a string which consists of a count and that number of bytes stored at the address to a signed double precision number using the current number base. If there is a point in the string its position is indicated in the user variable DPL . If the string cannot be converted an error message is given.
(NUMBER)	addr d	addr d	Converts ASCII characters stored at addr +1 to digits and accumulates them into the the double precision number d. Returns the address of the first unconvertable character. Used by NUMBER .
PAD		addr	Returns an address in the free memory area above the dictionary and a fixed offset from the dictionary pointer which can be used for the temporary storage of text.
QUERY			Accepts up to 80 characters from the keyboard terminated by return to the text input buffer and resets IN to zero.
SPACE			Outputs a space.
SPACES	n		Outputs n spaces.
?TERMINAL		f	If any key is being pressed returns a true flag; otherwise returns a false flag.
-TRAILING	n addr	n addr	Returns n which is the count of characters in a string stored at addr adjusted to eliminate trailing blanks. The address is returned unaltered.
TYPE	n addr		Outputs a string of n characters stored at addr.

WORD	c		Reads text from the input stream using c as a delimiter. Moves the text to the free space at the dictionary pointer with a character count in the first byte and followed by two or more spaces.

Number Formatting Operators

Word	Stack		Action
<#			Initiates formatted numeric output in which a double precision number is converted to text at PAD. The text includes the digits of the number.
#	d	d	Generates a digit from d by dividing by BASE leaving a quotient on the stack for further processing. The ASCII value of the digit is put into a string which is build up at PAD. Always generates a digit even if it is zero.
#S	d	d	Converts a number to a string of digits by repeating the action of # until the number on the stack is zero. Always generates at least one digit.
HOLD	c		Inserts in the string being built up at PAD the character whose ASCII value is on the stack.
SIGN			Inserts a minus sign in the string being built up at PAD if the third stack entry (immediately below the double precision number being converted) is negative.
#>	d	u addr	Completes number conversion by discarding the double precision number on the stack (which is usually zero by this stage) and leaving a count and address of the string at PAD. These can be used by TYPE to

output the string.

Vocabulary Control Words

Word	Stack	Action
CONTEXT	addr	User variable which contains the address of the vocabulary from which dictionary searches are started.
CURRENT	addr	User variable which contains the address of the vocabulary to which new dictionary entries will be linked.
DEFINITIONS		Makes the current vocabulary the same as the context vocabulary
FORTH		The name of the primary vocabulary which initially is the kernel or core language. New word definitions are automatically added to the FORTH vocabulary until other vocabularies are defined by the user.
FORGET		Discards the word whose name follows FORGET in the input line from the dictionary and all subsequent dictionary entries in dictionary order. The current and context vocabularies must be the same.
IMMEDIATE		Makes the latest entry in the current vocabulary an immediate word. Immediate words are executed and not compiled in colon word definitions making the compiling process self extending as well as the rest of the language.
LATEST	addr	Returns the name field address of the latest entry in the current vocabulary.

TASK	A word used to mark out a boundary in the dictionary between groups of word entries. The group, usually of words used in the same application, can be removed as a whole by FORGET TASK . TASK does not perform any active operation.
VLIST	Lists on the screen the names of the words of the context vocabulary. The listing can be interrupted from the keyboard.
VOC-LINK	User variable which contains the address of a field in the most recently defined vocabulary name word.
VOCABULARY	Defining word which is used to define vocabulary names. VOCABULARY EDITOR is used to define the word EDITOR which when used evokes the EDITOR vocabulary by placing its address in CONTEXT.

User Variables

Word	Stack	Action
SO	addr	Initial value of parameter stack pointer.
RO	addr	Initial value of return stack pointer.
TIB	addr	Address of terminal input buffer.
WIDTH	addr	Number of characters used to identify word names in the dictionary. The permitted range is 1 to 31.
WARNING	addr	Controls the treatment of error conditions. 0 Error message numbers only given.

		1 An error message relative to line 0 screen 4 drive 1 is printed.
		-1 (ABORT) is executed. (ABORT) may contain a user defined error procedure.
FENCE	addr	Marks the end of the protected dictionary; below this address the operation of FORGET is trapped.
DP	addr	The dictionary pointer is the address of the first free dictionary location.
VOC-LINK	addr	Contains the address of the last field of the most recently created vocabulary. The fields link vocabulary names.
BLK	addr	Contains the number of the block being interpreted unless the number is zero in which case the terminal buffer is interpreted.
IN	addr	Contains an offset from the start of the terminal input buffer or block being interpreted. Altered by each successive call of WORD.
OUT	addr	Incremented by each use of EMIT to output a character. May be used for output formatting.
SCR	addr	Contains the number of the screen most recently referenced by LIST.
OFFSET	addr	Contains a block offset used when the disc drives or other mass storage are accessed. Does not affect the behaviour of MESSAGE which always relates to line 0 screen 4 drive 1.

Word	Stack	Action
CONTEXT	addr	Contains the address of the vocabulary within which dictionary searches begin.
CURRENT	addr	Contains the address of the vocabulary to which new entries in the dictionary will be linked.
STATE	addr	Controls operation of text interpreter. If STATE contains a non zero value then words and numbers are compiled and not executed unless the words are immediate words.
BASE	addr	Contains current number base.
DPL	addr	Contains the number of digits to the right of the point in the last double precision number entered or -1 if the last number entered was single precision. May be used to hold the column position of the point in formatted number output.
FLD	addr	May be used to control field width in formatted number output.
CSP	addr	May be used to store parameter stack pointer position for error checking during compiling.
R#	addr	Contains the current position of the editing cursor.
HLD	addr	Contains a pointer to the current character position in the text at pad during number output.

System Constants

Word	Stack		Action
0 1 2 3		n	The numbers zero one two and three are in the dictionary and so available in any number base.
BL		n	Returns ASCII space 20 hex.
C/L		n	Characters per line.
FIRST		n	Leaves address of first block buffer.
LIMIT		n	Leaves the first address after that used by the last block buffer.
B/BUF		n	The number of bytes in each block buffer.
B/SCR		n	The number of buffers needed for each screen.

System Operators

Word	Stack		Action
'		addr	Tick is used in the sequence cccc and leaves the parameter field address of the word cccc which must be in a currently accessible vocabulary.
(			Together with) delineates a a comment which is ignored by the text interpreter.
DECIMAL			Sets the number base to decimal.
HEX			Sets the number base to hexadecimal or base 16.
EXECUTE	addr		Executes the word whose code field address is on the stack.
CFA	pfa	cfa	Converts the parameter field address of a word to the code field address.

Word	Stack		Action
LFA	pfa	lfa	Converts the parameter field address of a word to the link field address.
NFA	pfa	nfa	Converts the parameter field address of a word to the name field address.
PFA	nfa	pfa	Converts the name field address of a word to the parameter field address.
TRAVERSE	n addr	addr	Traverses the variable length name field of a word. If n = 1 converts the address of the length byte to that of the last letter. If n = -1 converts the address of the last letter to that of the length byte.
TOGGLE	b addr		Exclusive ORs the byte at the address with the bit mask b .
QUIT			Stops compilation, clears the return stack and returns control to the terminal without printing the message OK .
COLD			Restarts language clearing dictionary buffers and stacks allocating memory in accordance with cold start parameters.
WARM			Restarts language clearing buffers and stacks.
BYE			Exit to operating system or monitor.
+ORIGIN	n	addr	Converts the offset n to a memory address in the lowest part of memory used by the language. This area is reserved for certain cold start and system parameters.
P@	n	data	Reads port number n and leaves result on the stack.

P!	n data	Writes data (a single precision integer between 0 and 255) to port number n.
TASK		Marks the end of the kernel in the dictionary and performs no active operation.
NOOP		Forth non operation.
.CPU		Prints the name of the system CPU.
NULL		ASCII zero which is always the last word of interpretable text.
LIT	n	The run time procedure which pushes to the stack the 16 bit number following its address embedded in compiled code. Automatically used when numbers appear in colon definitions and other compiled text.
INTERPRET		Text interpreter which interprets the text it obtains from repeated calls to WORD. Return to the calling word is affected by the execution of NULL which is the ASCII zero always placed at the end of the text being interpreted.
(DO) (LOOP) (+LOOP) BACK BRANCH OBRANCH		The run time procedures which implement the conditional and looping structures. May be used only inside word definitions.
DIGIT (FIND) -FIND ENCLOSE		Internal procedures whose functions are available to the user through other words.
SMUDGE		Used in compilation to set and reset a bit in the length byte of each dictionary header. The smudge bit is used to make a word whose compiled code is incomplete inaccessible until compilation is completed without error.

Virtual Memory Operators

Word	Stack		Action
BLOCK	n	addr	Leaves the address of the buffer containing block n having read the block from tape if it is not already resident.
+BUF	addr	f addr	Using the address of the current buffer returns the address of the next buffer. The flag is false when the address returned is that in PREV .
BUFFER	n	addr	Obtains the next free buffer and assigns it to block n returning its address. The existing contents of the buffer are written to tape if necessary.
EMPTY-BUFFERS			Marks all buffers as empty. The contents are not written to tape.
FLUSH			Writes all updated buffers to tape.
INDEX	to from		Displays the first line of each screen in the range from to . These lines are usually used to describe the contents of each screen in a comment.
.LINE	scr line		Print a line of text accessed by line and screen number.
(LINE)	scr line	n addr	Returns the buffer address and number of characters in a line accessed by line and screen number.
LIST	n		Display the text of screen n . n is also stored in the user variable SCR .
LOAD	n		Interpret the text of screen n .

Word	Stack		Action
R/W	f blk addr		Reads from and writes to tape or other mass storage. The flag indicates the operation 0 write 1 read . The number blk specifies the block number and the address is that of the buffer assigned to the block.
WRITE-CHAR	n	f	Writes n to tape interface. Returns f=1 if error.
READ-CHAR	n	f n	Reads character from tape. f=1 if error occurred in which case n meaningless.
WRITE-HEAD	c a	f	Writes a header to tape c bytes long starting at addr a prefixed by 8 EA Hex chars. Returns f=1 if error occurs.
WRITE-BODY	a	f	Writes 1k block starting at addr a plus a checksum to tape. Returns f=1 if an error occurs.
FILE$	n	c a	Forms string at PAD in the form SCRN#nnn where n is the screen number returns a=PAD AND C=8.
READ-HEAD	c a	a1 f	Reads tape looking for file with name given by string starting at a with length c. Returns f=1 and no a1 if error or f=0 if OK with a1. If a1=0 then the file was found otherwise a1= the start of the string name of the file found.
READ-BODY	a	f	Reads 1k bytes and places them starting at addr a. Reads checksum & returns f=1 if error found.
BLOCK-FIND	n		Searches tape for SCRN#nnn. Aborts if an error found.
BLOCK-READ	n a		Reads block n from tape putting in buffer starting at address a.

BLOCK-WRITE n a		Writes block n to tape using data starting at address a.
TRIAD	n	Displays the text of three screens including screen n starting with a screen number evenly divisible by three. Also prints as a reference line line 15 of screen 4 .
UPDATE		Marks the most recently referenced block pointed to by PREV as altered. This block will be rewritten to tape when its buffer is reused or by FLUSH.
-->		Next screen continues interpretation with the next screen.
;S		Terminates interpretation of a screen.

Error Control Words

Word	Stack		Action
ERROR	n	BLK IN	Reports error number or message and restarts. Action depends on the value of the user variable WARNING. If WARNING = 0 then n is printed as a message number. If WARNING = 1 the text of line n relative to line 0 of screen 4 of drive 0 is printed. If WARNING = -1 then (ABORT) is executed. IN and BLK are left on the stack to locate the cause of the error.
(ABORT)			Executes ABORT . May be altered to execute user chosen error procedure.
ABORT			Clears stacks and returns control to terminal in execution mode. The system message is printed.

Word	Stack	Action
ID.	addr	Prints the name of a word from its name field address.
?ERROR	n f	Issue error message n only if the flag is true.
?COMP		Issue error message if not compiling.
?CSP		Issue error message if stack pointer differs from value saved in CSP .
?EXEC		Issue error message if not executing.
?LOADING		Issue error message if not loading.
?PAIRS	n1 n2	Issue error message if n1 and n2 are not equal. Used to detect incorrectly formed conditional constructs.
?STACK		Issue error message if the stack has underflowed or overflowed.
!CSP		Saves value of stack pointer in CSP. Used to detect compiling errors.
MESSAGE	n	Prints the text of line n relative to line 0 of screen 4 of drive 0 unless WARNING is zero in which case only the message number will be printed.

Editor Words

Word	Stack		Action
?NUM		ff or tf n	Attemps to convert the string at HERE preceded by a count to a single precision number. Returns the number plus true flag or a false flag.
ALTER	n		Moves the data at HERE to line n on the current screen (given by SCR). Trims the line to 64

Word	Stack	Action
		chs if it is longer.
EDIT	n	Edits screen n.
EDITOR	n	The name of the editor vocabulary
WHERE	n s	Prints line containing char number n in block s together with an arrow. Used when correcting program errors.